z/OS

Standard C++ Library Reference

Version 2 Release 1

SC14-7309-00

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 519.

This edition applies to IBM z/OS XL C/C++ compiler in IBM z/OS operating system Version 2 Release 1 (5650-ZOS) and to all subsequent releases until otherwise indicated in new editions. This edition replaces SC09-4949-05. Make sure that you use the correct edition for the level of the program listed above. Also, ensure that you apply all necessary PTFs for the program.

IBM welcomes your comments. You can send your comments to the following Internet address: compinfo@ca.ibm.com. Be sure to include your e-mail address if you want a reply. Include the title and order number of this book, and the page number or topic related to your comment. When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Chapter 1. Standard C++ Library Overview

The Standard C++ Library is supplied by IBM, and this manual is based on the Dinkum C++ Library and the *Dinkum C++ Library Reference*.

Use of this Dinkum C++ Library Reference is subject to limitations. See the Dinkumware Notices and the IBM Notices for detailed restrictions. Also, see the specific copyright notice at the bottom of this page.

A C++ program can call on a large number of functions from the **Dinkum C++ Library**, a conforming implementation of the **Standard C++ Library**. These functions perform essential services such as input and output. They also provide efficient implementations of frequently used operations. Numerous function and class definitions accompany these functions to help you to make better use of the library. Most of the information about the Standard C++ Library can be found in the descriptions of the **C++ library headers** that declare or define library entities for the program. The C++ library headers have two broader subdivisions, iostreams headers and STL headers.

The Standard C++ Library works in conjunction with the headers from the Standard C Library. For information about the Standard C Library, refer to the documentation that is supplied with the operating system.

A few special conventions are introduced into this document specifically for this particular implementation of the Standard C++ Library. Not all implementations support all the features described here. Hence, this implementation introduces macros, or alternative declarations, where necessary to provide reasonable substitutes for the capabilities required by the C++ Standard.

The Standard C++ Library is based on the C++03 standard and has not been updated to C++0x. The C++0x library functions will be updated in a future release.

Other information about the Standard C++ Library includes:

Multibyte Characters
How to convert between multibyte characters and wide characters.

Files and Streams
How to read and write data between the program and files.

Formatted Output
How to generate text under control of a format string.

Formatted Input
How to scan and parse text under control of a format string.

STL Conventions
How to read the descriptions of STL template classes and functions.

Containers
How to use an arbitrary STL container template class.

Copyright notice

Certain materials included or referred to in this document are copyright P.J. Plauger and/or Dinkumware, Ltd. or are based on materials that are copyright P.J.

Header files overview

The Standard C++ Library consists of 40 headers. Of these 40 headers, 15 constitute the **Standard Template Library**, or **STL**. 10 were added or updated with **C++ Technical Report 1**, or **TR1**. These are indicated below with the notations (STL) and (TR1):

<algorithm> — (STL) for defining numerous templates that implement useful algorithms
<array> — (TR1) for defining a fixed-size array with a container-like interface
<bitset> — for defining a template class that administers sets of bits
<complex> — for defining a template class that supports complex arithmetic
<deque> — (STL) for defining a template class that implements a deque container
<exception> — for defining several functions that control exception handling
<fstream> — for defining several iostreams template classes that manipulate exteral files
<functional> — (STL) / (TR1) for defining several templates that help construct predicates for the templates defined in <algorithm> and <numeric>
<iomanip> — for declaring several iostreams manipulators that take an argument
<ios> — for defining the template class that serves as the base for many iostreams classes
<iosfwd> — for declaring several iostreams template classes before they are necessarily defined
<iostream> — for declaring the iostreams objects that manipulate the standard streams
<istream> — for defining the template class that performs extractions
<iterator> — (STL) for defining several templates that help define and manipulate iterators
<limits> — for testing numeric type properties
<list> — (STL) for defining a template class that implements a list container
<locale> — for defining several classes and templates that control locale-specific behavior, as in the iostreams classes
<map> — (STL) for defining template classes that implement associative containers that map keys to values
<memory> — (STL) / (TR1) for defining templates that use reference counting to manage resources
<new> — for declaring several functions that allocate and free storage

<numeric> — (STL) for defining several templates that implement useful numeric functions
<ostream> — for defining the template class that performs insertions
<queue> — (STL) for defining a template class that implements a queue container
<random> — (TR1) for defining random number generators
<regex> — (TR1) for defining a template class to parse regular expressions and several template classes and functions to search text for matches to a regular expression object
<set> — (STL) for defining template classes that implement associative containers
<sstream> — for defining several iostreams template classes that manipulate string containers
<stack> — (STL) for defining a template class that implements a stack container
<stdexcept> — for defining several classes useful for reporting exceptions
<streambuf> — for defining template classes that buffer iostreams operations
<string> — for defining a template class that implements a string container
<strstream> — for defining several iostreams classes that manipulate in-memory character sequences
<typeinfo> — for defining class `type_info`, the result of the `typeid` operator
<type_traits> — (TR1) for accessing detailed type information at compile time to support generic programming
<tuple> — (TR1) for defining a template tuple whose instances hold objects of varying types
<unordered_map> — (STL) / (TR1) for defining template classes that implement unordered associative containers that map keys to values
<unordered_set> — (STL) / (TR1) for defining template classes that implement unordered associative containers
<utility> — (STL) / (TR1) for defining two tuple-like templates that provide information about the contents of instances of std::pair
<valarray> — for defining several classes and template classes that support value-oriented arrays
<vector> — (STL) for defining a template class that implements a vector container

Using C++ Library Headers

You include the contents of a standard header by naming it in an *include* directive, as in:

```
#include <iostream>  /* include I/O facilities */
```

You can include the standard headers in any order, a standard header more than once, or two or more standard headers that define the same macro or the same type. Do not include a standard header within a declaration. Do not define macros that have the same names as keywords before you include a standard header.

A C++ library header includes any other C++ library headers it needs to define needed types. (Always include explicitly any C++ library headers needed in a translation unit, however, lest you guess wrong about its actual dependencies.) A Standard C header never includes another standard header. A standard header declares or defines only the entities described for it in this document.

Every function in the library is declared in a standard header. Unlike in Standard C, the standard header never provides a masking macro, with the same name as the function, that masks the function declaration and achieves the same effect.

All names other than `operator delete` and `operator new` in the C++ library headers are defined in the **std** namespace, or in a namespace nested within the `std` namespace. Including a C++ library header does *not* introduce any library names into the current namespace. You refer to the name cin, for example, as `std::cin`. Alternatively, you can write the declaration:

```
using namespace std;
```

which promotes all library names into the current namespace. If you write this declaration immediately after all *include* directives, you can otherwise ignore namespace considerations in the remainder of the translation unit. Note that macro names are not subject to the rules for nesting namespaces.

Note that the C Standard headers behave mostly as if they include no namespace declarations. If you include, for example, <cstdlib>, you should call `std::abort()` to cause abnormal termination, but if you include <stdlib.h>, you should call `abort()`. (The C++ Standard is intentionally vague on this topic, so you should stick with just the usages described here for maximum portability.)

Unless specifically indicated otherwise, you may not define names in the `std` namespace, or in a namespace nested within the `std` namespace.

Standard C++ Library Conventions

The Standard C++ Library obeys much the same conventions as the Standard C Library, plus a few more outlined here.

An implementation has certain latitude in how it declares types and functions in the Standard C++ Library:

- Names of functions in the Standard C Library may have either **extern "C++"** or **extern "C"** linkage. Include the appropriate Standard C header rather than declare a library entity inline.
- A member function name in a library class may have additional function signatures over those listed in this document. You can be sure that a function call described here behaves as expected, but you cannot reliably take the address of a library member function. (The type may not be what you expect.)
- A library class may have undocumented (non-virtual) base classes. A class documented as derived from another class may, in fact, be derived from that class through other undocumented classes.
- A type defined as a synonym for some integer type may be the same as one of several different integer types.
- A **bitmask type** can be implemented as either an integer type or an enumeration. In either case, you can perform bitwise operations (such as AND and OR) on values of the same bitmask type. The *elements* `A` and `B` of a bitmask type are nonzero values such that `A & B` is zero.
- A library function that has no exception specification can throw an arbitrary exception, unless its definition clearly restricts such a possibility.

On the other hand, there are some restrictions you can count on:

- The Standard C Library uses no masking macros. Only specific function signatures are reserved, not the names of the functions themselves.
- A library function name outside a class will *not* have additional, undocumented, function signatures. You can reliably take its address.

- Base classes and member functions described as virtual are assuredly virtual, while those described as non-virtual are assuredly non-virtual.
- Two types defined by the C++ Library are always different unless this document explicitly suggests otherwise.
- Functions supplied by the library, including the default versions of replaceable functions, can throw *at most* those exceptions listed in any exception specification. No destructors supplied by the library throw exceptions. Functions in the Standard C Library may propagate an exception, as when qsort calls a comparison function that throws an exception, but they do not otherwise throw exceptions.

Iostreams Conventions

The **iostreams** headers support conversions between text and encoded forms, and input and output to external files: <fstream>, <iomanip>, <ios>, <iosfwd>, <iostream>, <istream>, <ostream>, <sstream>, <streambuf>, and <strstream>.

The simplest use of iostreams requires only that you include the header `<iostream>`. You can then extract values from cin, to read the standard input. The rules for doing so are outlined in the description of the class basic_istream. You can also insert values to cout, to write the standard output. The rules for doing so are outlined in the description of the class basic_ostream. Format control common to both extractors and insertors is managed by the class basic_ios. Manipulating this format information in the guise of extracting and inserting objects is the province of several manipulators.

You can perform the same iostreams operations on files that you open by name, using the classes declared in `<fstream>`. To convert between iostreams and objects of class basic_string, use the classes declared in `<sstream>`. And to do the same with C strings, use the classes declared in `<strstream>`.

The remaining headers provide support services, typically of direct interest to only the most advanced users of the iostreams classes.

C++ Program Startup and Termination

A C++ program performs the same operations as does a C program at program startup and at program termination, plus a few more outlined here.

Before the target environment calls the function `main`, and after it stores any constant initial values you specify in all objects that have static duration, the program executes any remaining constructors for such static objects. The order of execution is not specified between translation units, but you can nevertheless assume that some iostreams objects are properly initialized for use by these static constructors. These control text streams:

- cin — for standard input
- cout — for standard output
- cerr — for unbuffered standard error output
- clog — for buffered standard error output

You can also use these objects within the destructors called for static objects, during program termination.

As with C, returning from main or calling exit calls all functions registered with atexit in reverse order of registry. An exception thrown from such a registered function calls `terminate()`.

Chapter 2. Standard C++ Library Header Files

The Standard C++ Library can be categorized as follows:

- The Language Support Library
- The Diagnostics Library
- The General Utilities Library
- The Standard String Templates
- Localization Classes and Templates
- The Containers, Iterators and Algorithms Libraries (the Standard Template Library)
- The Standard Numerics Library
- The Standard Input/Output Library
- C++ Headers for the Standard C Library
- C++ Headers added with TR1

The Language Support Library The Language Support Library defines types and functions that will be used implicitly by C++ programs that employ such C++ language features as operators new and delete, exception handling and runtime type information (RTTI).

Standard C++ header	Equivalent in previous versions
<exception>	no equivalent
<limits>	no equivalent
<new>	<new.h>
<typeinfo>	no equivalent

The Diagnostics Library The Diagnostics Library is used to detect and report error conditions in C++ programs.

Standard C++ header	Equivalent in previous versions
<stdexcept>	no equivalent

The General Utilities Library The General Utilities Library is used by other components of the Standard C++ Library, especially the Containers, Iterators and Algorithms Libraries (the Standard Template Library).

Standard C++ header	Equivalent in previous versions
<utility>	no equivalent
<functional>	no equivalent
<memory>	no equivalent

The Standard String Templates The Strings Library is a facility for the manipulation of character sequences.

Standard C++ header	Equivalent in previous versions
<string>	no equivalent

Localization Classes and Templates The Localization Library permits a C++ program to address the cultural differences of its various users.

Standard C++ header	Equivalent in previous versions
<locale>	no equivalent

The Containers, Iterators and Algorithms Libraries (the Standard Template Library) The Standard Template Library (STL) is a facility for the management and manipulation of collections of objects.

Standard C++ header	Equivalent in previous versions
<algorithm>	no equivalent
<bitset>	no equivalent
<deque>	no equivalent
<iterator>	no equivalent
<list>	no equivalent
<map>	no equivalent
<queue>	no equivalent
<set>	no equivalent
<stack>	no equivalent
<unordered_map>	no equivalent
<unordered set>	no equivalent
<vector>	no equivalent

The Standard Numerics Library The Numerics Library is a facility for performing seminumerical operations.

Users who require library facilities for complex arithmetic but wish to maintain compatibility with older compilers may use the compatibility complex numbers library whose types are defined in the non-standard header file <complex.h>. Although the header files <complex> and <complex.h> are similar in purpose, they are mutually incompatible.

Standard C++ header	Equivalent in previous versions
<complex>	no equivalent
<numeric>	no equivalent
<valarray>	no equivalent

The Standard Input/Output Library The standard iostreams library differs from the compatibility iostreams in a number of important respects. To maintain compatibility between such a product and VisualAge® C++ Version 5.0 or z/OS® C/C++ Version 1.2, use instead the compatibility iostreams library.

Standard C++ header	Equivalent in previous versions

<fstream>	no equivalent
<iomanip>	no equivalent
<ios>	no equivalent
<iosfwd>	no equivalent
<iostream>	no equivalent
<istream>	no equivalent
<ostream>	no equivalent
<streambuf>	no equivalent
<sstream>	no equivalent
<strstream>	no equivalent

C++ Headers for the Standard C Library The 1990 C International Standard specifies 18 headers which must be provided by a conforming hosted implementation. The name of each of these headers is of the form *name*.h. The C++ Standard Library includes the 1990 C Standard Library and, hence, includes these 18 headers. Additionally, for each of the 18 headers specified by the 1990 C International Standard, the C++ standard specifies a corresponding header that is functionally equivalent to its C library counterpart, but which locates all of the declarations that it contains within the std namespace. The name of each of these C++ headers is of the form c*name*, where *name* is the string that results when the ".h" extension is removed from the name of the equivalent C Standard Library header. For example, the header files <stdlib.h> and <cstdlib> are both provided by the C++ Standard Library and are equivalent in function, with the exception that all declarations in <cstdlib> are located within the `std` namespace.

Standard C++ Header	**Corresponding Standard C & C++ Header**
<cassert>	<assert.h>
<cctype>	<ctype.h>
<cerrno>	<errno.h>
<cfloat>	<float.h>
<ciso646>	<iso646.h>
<climits>	<limits.h>
<clocale>	<locale.h>
<cmath>	<math.h>
<csetjmp>	<setjmp.h>
<csignal>	<signal.h>
<cstdarg>	<stdarg.h>
<cstddef>	<stddef.h>
<cstdio>	<stdio.h>
<cstdlib>	<stdlib.h>
<cstring>	<string.h>
<ctime>	<time.h>
<cwchar>	<wchar.h>
<cwctype>	<wctype.h>

C++ Headers added with TR1: The following headers are added with TR1.

Standard C++ Header
<array>
<random>
<regex>
<type_traits>
<tuple>
<unordered_map>
<unordered_set>

<algorithm>

Description

Include the STL standard header **<algorithm>** to define numerous template functions that perform useful algorithms. The descriptions that follow make extensive use of common template parameter names or prefixes to indicate the least powerful category of iterator permitted as an actual argument type:

- **OutIt —** to indicate an output iterator
- **InIt —** to indicate an input iterator
- **FwdIt —** to indicate a forward iterator
- **BidIt —** to indicate a bidirectional iterator
- **RanIt —** to indicate a random-access iterator

The descriptions of these templates employ a number of conventions common to all algorithms.

Synopsis

```
namespace std {
template<class InIt, class Fun>
    Fun for_each(InIt first, InIt last, Fun f);
template<class InIt, class T>
    InIt find(InIt first, InIt last, const T& val);
template<class InIt, class Pred>
    InIt find_if(InIt first, InIt last, Pred pr);
template<class FwdIt1, class FwdIt2>
    FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
template<class FwdIt1, class FwdIt2>
    FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
template<class FwdIt>
    FwdIt adjacent_find(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);
template<class InIt, class T, class Dist>
    typename iterator_traits<InIt>::difference_type
        count(InIt first, InIt last,
```

```
            const T& val);
template<class InIt, class Pred, class Dist>
    typename iterator_traits<InIt>::difference_type
        count_if(InIt first, InIt last,
            Pred pr);
template<class InIt1, class InIt2>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
        InIt2 x);
template<class InIt1, class InIt2, class Pred>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
        InIt2 x, Pred pr);
template<class InIt1, class InIt2>
    bool equal(InIt1 first, InIt1 last, InIt2 x);
template<class InIt1, class InIt2, class Pred>
    bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);
template<class FwdIt1, class FwdIt2>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
template<class FwdIt, class Dist, class T>
    FwdIt search_n(FwdIt first, FwdIt last,
        Dist n, const T& val);
template<class FwdIt, class Dist, class T, class Pred>
    FwdIt search_n(FwdIt first, FwdIt last,
        Dist n, const T& val, Pred pr);
template<class InIt, class OutIt>
    OutIt copy(InIt first, InIt last, OutIt x);
template<class BidIt1, class BidIt2>
    BidIt2 copy_backward(BidIt1 first, BidIt1 last,
        BidIt2 x);
template<class T>
    void swap(T& x, T& y);
template<class FwdIt1, class FwdIt2>
    FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last,
        FwdIt2 x);
template<class FwdIt1, class FwdIt2>
    void iter_swap(FwdIt1 x, FwdIt2 y);
template<class InIt, class OutIt, class Unop>
    OutIt transform(InIt first, InIt last, OutIt x,
        Unop uop);
template<class InIt1, class InIt2, class OutIt,
    class Binop>
    OutIt transform(InIt1 first1, InIt1 last1,
        InIt2 first2, OutIt x, Binop bop);

template<class FwdIt, class T>
    void replace(FwdIt first, FwdIt last,
        const T& vold, const T& vnew);
template<class FwdIt, class Pred, class T>
    void replace_if(FwdIt first, FwdIt last,
        Pred pr, const T& val);
template<class InIt, class OutIt, class T>
    OutIt replace_copy(InIt first, InIt last, OutIt x,
        const T& vold, const T& vnew);
template<class InIt, class OutIt, class Pred, class T>
    OutIt replace_copy_if(InIt first, InIt last, OutIt x,
        Pred pr, const T& val);
template<class FwdIt, class T>
    void fill(FwdIt first, FwdIt last, const T& x);
template<class OutIt, class Size, class T>
    void fill_n(OutIt first, Size n, const T& x);
template<class FwdIt, class Gen>
    void generate(FwdIt first, FwdIt last, Gen g);
template<class OutIt, class Pred, class Gen>
    void generate_n(OutIt first, Dist n, Gen g);
template<class FwdIt, class T>
```

```
    FwdIt remove(FwdIt first, FwdIt last, const T& val);
template<class FwdIt, class Pred>
    FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
template<class InIt, class OutIt, class T>
    OutIt remove_copy(InIt first, InIt last, OutIt x,
        const T& val);
template<class InIt, class OutIt, class Pred>
    OutIt remove_copy_if(InIt first, InIt last, OutIt x,
        Pred pr);
template<class FwdIt>
    FwdIt unique(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt unique(FwdIt first, FwdIt last, Pred pr);
template<class InIt, class OutIt>
    OutIt unique_copy(InIt first, InIt last, OutIt x);
template<class InIt, class OutIt, class Pred>
    OutIt unique_copy(InIt first, InIt last, OutIt x,
        Pred pr);
template<class BidIt>
    void reverse(BidIt first, BidIt last);
template<class BidIt, class OutIt>
    OutIt reverse_copy(BidIt first, BidIt last, OutIt x);
template<class FwdIt>
    void rotate(FwdIt first, FwdIt middle, FwdIt last);
template<class FwdIt, class OutIt>
    OutIt rotate_copy(FwdIt first, FwdIt middle,
        FwdIt last, OutIt x);
template<class RanIt>
    void random_shuffle(RanIt first, RanIt last);
template<class RanIt, class Fun>
    void random_shuffle(RanIt first, RanIt last, Fun& f);
template<class BidIt, class Pred>
    BidIt partition(BidIt first, BidIt last, Pred pr);
template<class BidIt, class Pred>
    BidIt stable_partition(BidIt first, BidIt last,
        Pred pr);
template<class RanIt>
    void sort(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort(RanIt first, RanIt last, Pred pr);
template<class BidIt>
    void stable_sort(BidIt first, BidIt last);
template<class BidIt, class Pred>
    void stable_sort(BidIt first, BidIt last, Pred pr);
template<class RanIt>
    void partial_sort(RanIt first, RanIt middle,
        RanIt last);
template<class RanIt, class Pred>
    void partial_sort(RanIt first, RanIt middle,
        RanIt last, Pred pr);
template<class InIt, class RanIt>
    RanIt partial_sort_copy(InIt first1, InIt last1,
        RanIt first2, RanIt last2);
template<class InIt, class RanIt, class Pred>
    RanIt partial_sort_copy(InIt first1, InIt last1,
        RanIt first2, RanIt last2, Pred pr);

template<class RanIt>
    void nth_element(RanIt first, RanIt nth, RanIt last);
template<class RanIt, class Pred>
    void nth_element(RanIt first, RanIt nth, RanIt last,
        Pred pr);
template<class FwdIt, class T>
    FwdIt lower_bound(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt lower_bound(FwdIt first, FwdIt last,
        const T& val, Pred pr);
```

```
template<class FwdIt, class T>
    FwdIt upper_bound(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt upper_bound(FwdIt first, FwdIt last,
        const T& val, Pred pr);
template<class FwdIt, class T>
    pair<FwdIt, FwdIt> equal_range(FwdIt first,
        FwdIt last, const T& val);
template<class FwdIt, class T, class Pred>
    pair<FwdIt, FwdIt> equal_range(FwdIt first,
        FwdIt last, const T& val, Pred pr);
template<class FwdIt, class T>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class BidIt>
    void inplace_merge(BidIt first, BidIt middle,
        BidIt last);
template<class BidIt, class Pred>
    void inplace_merge(BidIt first, BidIt middle,
        BidIt last, Pred pr);
template<class InIt1, class InIt2>
    bool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_symmetric_difference(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_symmetric_difference(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2, OutIt x,
            Pred pr);
template<class RanIt>
```

```
    void push_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void push_heap(RanIt first, RanIt last, Pred pr);
template<class RanIt>
    void pop_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void pop_heap(RanIt first, RanIt last, Pred pr);
template<class RanIt>
    void make_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void make_heap(RanIt first, RanIt last, Pred pr);
template<class RanIt>
    void sort_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort_heap(RanIt first, RanIt last, Pred pr);
template<class T>
    const T& max(const T& x, const T& y);
template<class T, class Pred>
    const T& max(const T& x, const T& y, Pred pr);
template<class T>
    const T& min(const T& x, const T& y);
template<class T, class Pred>
    const T& min(const T& x, const T& y, Pred pr);
template<class FwdIt>
    FwdIt max_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt max_element(FwdIt first, FwdIt last, Pred pr);
template<class FwdIt>
    FwdIt min_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt min_element(FwdIt first, FwdIt last, Pred pr);
template<class InIt1, class InIt2>
    bool lexicographical_compare(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool lexicographical_compare(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2, Pred pr);
template<class BidIt>
    bool next_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool next_permutation(BidIt first, BidIt last,
        Pred pr);
template<class BidIt>
    bool prev_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool prev_permutation(BidIt first, BidIt last,
        Pred pr);
    }
```

Functions

adjacent_find

```
template<class FwdIt>
    FwdIt adjacent_find(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);
```

The first template function determines the lowest `N` in the range `[0, last - first)` for which `N + 1 != last - first` and the predicate `*(first + N) == *(first + N + 1)` is true. Here, `operator==` must impose an equivalence relationship between its operands. It then returns `first + N`. If no such value exists, the function returns `last`. It evaluates the predicate exactly `N + 1` times.

The second template function behaves the same, except that the predicate is pr(*(first + N), *(first + N + 1)).

binary_search

```
template<class FwdIt, class T>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    bool binary_search(FwdIt first, FwdIt last,
        const T& val, Pred pr);
```

The first template function determines whether a value of N exists in the range [0, last - first) for which *(first + N) has equivalent ordering to val, where the elements designated by iterators in the range [first, last) form a sequence ordered by operator<. If so, the function returns true. If no such value exists, it returns false.

If FwdIt is a random-access iterator type, the function evaluates the ordering predicate X < Y at most ceil(log(last - first)) + 2 times. Otherwise, the function evaluates the predicate a number of times proportional to last - first.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

copy

```
template<class InIt, class OutIt>
    OutIt copy(InIt first, InIt last, OutIt x);
```

The template function evaluates *(x + N) = *(first + N)) once for each N in the range [0, last - first), for strictly increasing values of N beginning with the lowest value. It then returns x + N. If x and first designate regions of storage, x must not be in the range [first, last).

copy_backward

```
template<class BidIt1, class BidIt2>
    BidIt2 copy_backward(BidIt1 first, BidIt1 last,
        BidIt2 x);
```

The template function evaluates *(x - N - 1) = *(last - N - 1)) once for each N in the range [0, last - first), for strictly decreasing values of N beginning with the highest value. It then returns x - (last - first). If x and first designate regions of storage, x must not be in the range [first, last).

count

```
template<class InIt, class T>
    typename iterator_traits<InIt>::difference_type
        count(InIt first, InIt last, const T& val);
```

The template function sets a count n to zero. It then executes ++n for each N in the range [0, last - first) for which the predicate *(first + N) == val is true. Here, operator== must impose an equivalence relationship between its operands. The function returns n. It evaluates the predicate exactly last - first times.

count_if

```
template<class InIt, class Pred, class Dist>
    typename iterator_traits<InIt>::difference_type
        count_if(InIt first, InIt last,
            Pred pr);
```

The template function sets a count n to zero. It then executes ++n for each N in the range [0, last - first) for which the predicate pr(*(first + N)) is true. The function returns n. It evaluates the predicate exactly last - first times.

equal

```
template<class InIt1, class InIt2>
    bool equal(InIt1 first, InIt1 last, InIt2 x);
template<class InIt1, class InIt2, class Pred>
    bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);
```

The first template function returns true only if, for each N in the range [0, last1 - first1), the predicate *(first1 + N) == *(first2 + N) is true. Here, operator== must impose an equivalence relationship between its operands. The function evaluates the predicate at most once for each N.

The second template function behaves the same, except that the predicate is pr(*(first1 + N), *(first2 + N)).

equal_range

```
template<class FwdIt, class T>
    pair<FwdIt, FwdIt> equal_range(FwdIt first,
        FwdIt last, const T& val);
template<class FwdIt, class T, class Pred>
    pair<FwdIt, FwdIt> equal_range(FwdIt first,
        FwdIt last, const T& val, Pred pr);
```

The first template function effectively returns pair(lower_bound(first, last, val), upper_bound(first, last, val)), where the elements designated by iterators in the range [first, last) form a sequence ordered by operator<. Thus, the function determines the largest range of positions over which val can be inserted in the sequence and still preserve its ordering.

If FwdIt is a random-access iterator type, the function evaluates the ordering predicate X < Y at most ceil(2 * log(last - first)) + 1. Otherwise, the function evaluates the predicate a number of times proportional to last - first.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

fill

```
template<class FwdIt, class T>
    void fill(FwdIt first, FwdIt last, const T& x);
```

The template function evaluates *(first + N) = x once for each N in the range [0, last - first).

fill_n

```
template<class OutIt, class Size, class T>
    void fill_n(OutIt first, Size n, const T& x);
```

The template function evaluates *(first + N) = x once for each N in the range [0, n).

find

```
template<class InIt, class T>
    InIt find(InIt first, InIt last, const T& val);
```

The template function determines the lowest value of `N` in the range `[0, last - first)` for which the predicate `*(first + N) == val` is true. Here, `operator==` must impose an equivalence relationship between its operands. It then returns `first + N`. If no such value exists, the function returns `last`. It evaluates the predicate at most once for each `N`.

find_end

```
template<class FwdIt1, class FwdIt2>
    FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
```

The first template function determines the highest value of `N` in the range `[0, last1 - first1 - (last2 - first2))` such that for each `M` in the range `[0, last2 - first2)`, the predicate `*(first1 + N + M) == *(first2 + N + M)` is true. Here, `operator==` must impose an equivalence relationship between its operands. It then returns `first1 + N`. If no such value exists, the function returns `last1`. It evaluates the predicate at most `(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)` times.

The second template function behaves the same, except that the predicate is `pr(*(first1 + N + M), *(first2 + N + M))`.

find_first_of

```
template<class FwdIt1, class FwdIt2>
    FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
```

The first template function determines the lowest value of `N` in the range `[0, last1 - first1)` such that for some `M` in the range `[0, last2 - first2)`, the predicate `*(first1 + N) == *(first2 + M)` is true. Here, `operator==` must impose an equivalence relationship between its operands. It then returns `first1 + N`. If no such value exists, the function returns `last1`. It evaluates the predicate at most `(last1 - first1) * (last2 - first2)` times.

The second template function behaves the same, except that the predicate is `pr(*(first1 + N), *(first2 + M))`.

find_if

```
template<class InIt, class Pred>
    InIt find_if(InIt first, InIt last, Pred pr);
```

The template function determines the lowest value of `N` in the range `[0, last - first)` for which the predicate `pred(*(first + N))` is true. It then returns `first + N`. If no such value exists, the function returns `last`. It evaluates the predicate at most once for each `N`.

for_each

```
template<class InIt, class Fun>
    Fun for_each(InIt first, InIt last, Fun f);
```

The template function evaluates `f(*(first + N))` once for each `N` in the range `[0, last - first)`. It then returns `f`.

generate

```
template<class FwdIt, class Gen>
    void generate(FwdIt first, FwdIt last, Gen g);
```

The template function evaluates `*(first + N) = g()` once for each `N` in the range `[0, last - first)`.

generate_n

```
template<class OutIt, class Pred, class Gen>
    void generate_n(OutIt first, Dist n, Gen g);
```

The template function evaluates `*(first + N) = g()` once for each `N` in the range `[0, n)`.

includes

```
template<class InIt1, class InIt2>
    bool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, Pred pr);
```

The first template function determines whether a value of `N` exists in the range `[0, last2 - first2)` such that, for each `M` in the range `[0, last1 - first1)`, `*(first + M)` and `*(first + N)` do not have equivalent ordering, where the elements designated by iterators in the ranges `[first1, last1)` and `[first2, last2)` each form a sequence ordered by `operator<`. If so, the function returns false. If no such value exists, it returns true. Thus, the function determines whether the ordered sequence designated by iterators in the range `[first2, last2)` all have equivalent ordering with some element designated by iterators in the range `[first1, last1)`.

The function evaluates the predicate at most `2 * ((last1 - first1) + (last2 - first2)) - 1` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

inplace_merge

```
template<class BidIt>
    void inplace_merge(BidIt first, BidIt middle,
        BidIt last);
template<class BidIt, class Pred>
    void inplace_merge(BidIt first, BidIt middle,
        BidIt last, Pred pr);
```

The first template function reorders the sequences designated by iterators in the ranges `[first, middle)` and `[middle, last)`, each ordered by `operator<`, to form a merged sequence of length `last - first` beginning at `first` also ordered by `operator<`. The merge occurs without altering the relative order of elements within either original sequence. Moreover, for any two elements from different original sequences that have equivalent ordering, the element from the ordered range `[first, middle)` precedes the other.

The function evaluates the ordering predicate `X < Y` at most `ceil((last - first) * log(last - first))` times. (Given enough temporary storage, it can evaluate the predicate at most `(last - first) - 1` times.)

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

iter_swap

```
template<class FwdIt1, class FwdIt2>
    void iter_swap(FwdIt1 x, FwdIt2 y);
```

The template function leaves the value originally stored in *y subsequently stored in *x, and the value originally stored in *x subsequently stored in *y.

lexicographical_compare

```
template<class InIt1, class InIt2>
    bool lexicographical_compare(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool lexicographical_compare(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2, Pred pr);
```

The first template function determines K, the number of elements to compare as the smaller of last1 - first1 and last2 - first2. It then determines the lowest value of N in the range [0, K) for which *(first1 + N) and *(first2 + N) do not have equivalent ordering. If no such value exists, the function returns true only if K < (last2 - first2). Otherwise, it returns true only if *(first1 + N) < *(first2 + N). Thus, the function returns true only if the sequence designated by iterators in the range [first1, last1) is lexicographically less than the other sequence.

The function evaluates the ordering predicate X < Y at most 2 * K times.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

lower_bound

```
template<class FwdIt, class T>
    FwdIt lower_bound(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt lower_bound(FwdIt first, FwdIt last,
        const T& val, Pred pr);
```

The first template function determines the highest value of N in the range (0, last - first] such that, for each M in the range [0, N) the predicate *(first + M) < val is true, where the elements designated by iterators in the range [first, last) form a sequence ordered by operator<. It then returns first + N. Thus, the function determines the lowest position before which val can be inserted in the sequence and still preserve its ordering.

If FwdIt is a random-access iterator type, the function evaluates the ordering predicate X < Y at most ceil(log(last - first)) + 1 times. Otherwise, the function evaluates the predicate a number of times proportional to last - first.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

make_heap

```
template<class RanIt>
    void make_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void make_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range [first, last) to form a heap ordered by operator<.

The function evaluates the ordering predicate X < Y at most 3 * (last - first) times.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

max

```
template<class T>
    const T& max(const T& x, const T& y);
template<class T, class Pred>
    const T& max(const T& x, const T& y, Pred pr);
```

The first template function returns y if x < y. Otherwise it returns x. T need supply only a single-argument constructor and a destructor.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

max_element

```
template<class FwdIt>
    FwdIt max_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt max_element(FwdIt first, FwdIt last, Pred pr);
```

The first template function determines the lowest value of N in the range [0, last - first) such that, for each M in the range [0, last - first) the predicate *(first + N) < *(first + M) is false. It then returns first + N. Thus, the function determines the lowest position that contains the largest value in the sequence.

The function evaluates the ordering predicate X < Y exactly max((last - first) - 1, 0) times.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

merge

```
template<class InIt1, class InIt2, class OutIt>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function determines K, the number of elements to copy as (last1 - first1) + (last2 - first2). It then alternately copies two sequences, designated by iterators in the ranges [first1, last1) and [first2, last2) and each ordered by operator<, to form a merged sequence of length K beginning at x, also ordered by operator<. The function then returns x + K.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for any two elements from different sequences that have equivalent ordering, the element from the ordered range [first1, last1) precedes the other. Thus, the function merges two ordered sequences to form another ordered sequence.

If x and first1 designate regions of storage, the range [x, x + K) must not overlap the range [first1, last1). If x and first2 designate regions of storage, the range [x, x + K) must not overlap the range [first2, last2). The function evaluates the ordering predicate X < Y at most K - 1 times.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

min

```
template<class T>
    const T& min(const T& x, const T& y);
template<class T, class Pred>
    const T& min(const T& x, const T& y, Pred pr);
```

The first template function returns y if y < x. Otherwise it returns x. T need supply only a single-argument constructor and a destructor.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

min_element

```
template<class FwdIt>
    FwdIt min_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt min_element(FwdIt first, FwdIt last, Pred pr);
```

The first template function determines the lowest value of N in the range [0, last - first) such that, for each M in the range [0, last - first) the predicate *(first + M) < *(first + N) is false. It then returns first + N. Thus, the function determines the lowest position that contains the smallest value in the sequence.

The function evaluates the ordering predicate X < Y exactly max((last - first) - 1, 0) times.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

mismatch

```
template<class InIt1, class InIt2>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
        InIt2 x);
template<class InIt1, class InIt2, class Pred>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
        InIt2 x, Pred pr);
```

The first template function determines the lowest value of N in the range [0, last1 - first1) for which the predicate !(*(first1 + N) == *(first2 + N)) is true. Here, operator== must impose an equivalence relationship between its operands. It then returns pair(first1 + N, first2 + N). If no such value exists, N has the value last1 - first1. The function evaluates the predicate at most once for each N.

The second template function behaves the same, except that the predicate is pr(*(first1 + N), *(first2 + N)).

next_permutation

```
template<class BidIt>
    bool next_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool next_permutation(BidIt first, BidIt last,
        Pred pr);
```

The first template function determines a repeating sequence of permutations, whose initial permutation occurs when the sequence designated by iterators in the range `[first, last)` is ordered by `operator<`. (The elements are sorted in *ascending* order.) It then reorders the elements in the sequence, by evaluating `swap(X, Y)` for the elements `X` and `Y` zero or more times, to form the next permutation. The function returns true only if the resulting sequence is not the initial permutation. Otherwise, the resultant sequence is the one next larger lexicographically than the original sequence. No two elements may have equivalent ordering.

The function evaluates `swap(X, Y)` at most `(last - first) / 2`.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

nth_element

```
template<class RanIt>
    void nth_element(RanIt first, RanIt nth, RanIt last);
template<class RanIt, class Pred>
    void nth_element(RanIt first, RanIt nth, RanIt last,
        Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` such that for each `N` in the range `[0, nth - first)` and for each `M` in the range `[nth - first, last - first)` the predicate `!(*(first + M) < *(first + N))` is true. Moreover, for `N` equal to `nth - first` and for each `M` in the range `(nth - first, last - first)` the predicate `!(*(first + M) < *(first + N))` is true. Thus, if `nth != last` the element `*nth` is in its proper position if elements of the entire sequence were sorted in *ascending* order, ordered by `operator<`. Any elements before this one belong before it in the sort sequence, and any elements after it belong after it.

The function evaluates the ordering predicate `X < Y` a number of times proportional to `last - first`, on average.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

partial_sort

```
template<class RanIt>
    void partial_sort(RanIt first, RanIt middle,
        RanIt last);
template<class RanIt, class Pred>
    void partial_sort(RanIt first, RanIt middle,
        RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` such that for each `N` in the range `[0, middle - first)` and for each `M` in the range `(N, last - first)` the predicate `!(*(first + M) < *(first + N))` is true. Thus, the smallest `middle - first` elements of the entire sequence are sorted in *ascending* order, ordered by `operator<`. The order of the remaining elements is otherwise unspecified.

The function evaluates the ordering predicate `X < Y` at most `ceil((last - first) * log(middle - first))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

partial_sort_copy

```
template<class InIt, class RanIt>
    RanIt partial_sort_copy(InIt first1, InIt last1,
        RanIt first2, RanIt last2);
template<class InIt, class RanIt, class Pred>
    RanIt partial_sort_copy(InIt first1, InIt last1,
        RanIt first2, RanIt last2, Pred pr);
```

The first template function determines `K`, the number of elements to copy as the smaller of `last1 - first1` and `last2 - first2`. It then copies and reorders `K` of the sequence designated by iterators in the range `[first1, last1)` such that the `K` elements copied to `first2` are ordered by `operator<`. Moreover, for each `N` in the range `[0, K)` and for each `M` in the range `(0, last1 - first1)` corresponding to an uncopied element, the predicate `!(*(first2 + M) < *(first1 + N))` is true. Thus, the smallest `K` elements of the entire sequence designated by iterators in the range `[first1, last1)` are copied and sorted in *ascending* order to the range `[first2, first2 + K)`.

The function evaluates the ordering predicate `X < Y` at most `ceil((last - first) * log(K))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

partition

```
template<class BidIt, class Pred>
    BidIt partition(BidIt first, BidIt last, Pred pr);
```

The template function reorders the sequence designated by iterators in the range `[first, last)` and determines the value `K` such that for each `N` in the range `[0, K)` the predicate `pr(*(first + N))` is true, and for each `N` in the range `[K, last - first)` the predicate `pr(*(first + N))` is false. The function then returns `first + K`.

The predicate must not alter its operand. The function evaluates `pr(*(first + N))` exactly `last - first` times, and swaps at most `(last - first) / 2` pairs of elements.

pop_heap

```
template<class RanIt>
    void pop_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void pop_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a new heap, ordered by `operator<` and designated by iterators in the range `[first, last - 1)`, leaving the original element at `*first` subsequently at `*(last - 1)`. The original sequence must designate an existing heap, also ordered by `operator<`. Thus, `first != last` must be true and `*(last - 1)` is the element to remove from (pop off) the heap.

The function evaluates the ordering predicate `X < Y` at most `ceil(2 * log(last - first))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

prev_permutation

```
template<class BidIt>
    bool prev_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool prev_permutation(BidIt first, BidIt last,
        Pred pr);
```

The first template function determines a repeating sequence of permutations, whose initial permutation occurs when the sequence designated by iterators in the range `[first, last)` is the *reverse* of one ordered by `operator<`. (The elements are sorted in *descending* order.) It then reorders the elements in the sequence, by evaluating `swap(X, Y)` for the elements `X` and `Y` zero or more times, to form the next permutation. The function returns true only if the resulting sequence is not the initial permutation. Otherwise, the resultant sequence is the one next smaller lexicographically than the original sequence. No two elements may have equivalent ordering.

The function evaluates `swap(X, Y)` at most `(last - first) / 2`.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

push_heap

```
template<class RanIt>
    void push_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void push_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a new heap ordered by `operator<`. Iterators in the range `[first, last - 1)` must designate an existing heap, also ordered by `operator<`. Thus, `first != last` must be true and `*(last - 1)` is the element to add to (push on) the heap.

The function evaluates the ordering predicate `X < Y` at most `ceil(log(last - first))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

random_shuffle

```
template<class RanIt>
    void random_shuffle(RanIt first, RanIt last);
template<class RanIt, class Fun>
    void random_shuffle(RanIt first, RanIt last, Fun& f);
```

The first template function evaluates `swap(*(first + N), *(first + M))` once for each `N` in the range `[1, last - first)`, where `M` is a value from some uniform random distribution over the range `[0, N)`. Thus, the function randomly shuffles the order of elements in the sequence.

The second template function behaves the same, except that M is (Dist)f((Dist)N), where Dist is a type convertible to iterator_traits :: difference_type.

remove

```
template<class FwdIt, class T>
    FwdIt remove(FwdIt first, FwdIt last, const T& val);
```

The template function effectively assigns first to X, then executes the statement:

```
if (!(*(first + N) == val))
    *X++ = *(first + N);
```

once for each N in the range [0, last - first). Here, operator== must impose an equivalence relationship between its operands. It then returns X. Thus, the function removes from the sequence all elements for which the predicate *(first + N) == val is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

remove_copy

```
template<class InIt, class OutIt, class T>
    OutIt remove_copy(InIt first, InIt last, OutIt x,
        const T& val);
```

The template function effectively executes the statement:

```
if (!(*(first + N) == val))
    *x++ = *(first + N);
```

once for each N in the range [0, last - first). Here, operator== must impose an equivalence relationship between its operands. It then returns x. Thus, the function removes from the sequence all elements for which the predicate *(first + N) == val is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

If x and first designate regions of storage, the range [x, x + (last - first)) must not overlap the range [first, last).

remove_copy_if

```
template<class InIt, class OutIt, class Pred>
    OutIt remove_copy_if(InIt first, InIt last, OutIt x,
        Pred pr);
```

The template function effectively executes the statement:

```
if (!pr(*(first + N)))
    *x++ = *(first + N);
```

once for each N in the range [0, last - first). It then returns x. Thus, the function removes from the sequence all elements for which the predicate pr(*(first + N)) is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

If x and first designate regions of storage, the range [x, x + (last - first)) must not overlap the range [first, last).

remove_if

```
template<class FwdIt, class Pred>
    FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
```

The template function effectively assigns first to X, then executes the statement:

```
if (!pr(*(first + N)))
    *X++ = *(first + N);
```

once for each N in the range [0, last - first). It then returns X. Thus, the function removes from the sequence all elements for which the predicate pr(*(first + N)) is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

replace

```
template<caass FwdIt, class T>
    void replace(FwdIt first, FwdIt last,
        const T& vold, const T& vnew);
```

The template function executes the statement:

```
if (*(first + N) == vold)
    *(first + N) = vnew;
```

once for each N in the range [0, last - first). Here, operator== must impose an equivalence relationship between its operands.

replace_copy

```
template<class InIt, class OutIt, class T>
    OutIt replace_copy(InIt first, InIt last, OutIt x,
        const T& vold, const T& vnew);
```

The template function executes the statement:

```
if (*(first + N) == vold)
    *(x + N) = vnew;
else
    *(x + N) = *(first + N)
```

once for each N in the range [0, last - first). Here, operator== must impose an equivalence relationship between its operands.

If x and first designate regions of storage, the range [x, x + (last - first)) must not overlap the range [first, last).

replace_copy_if

```
template<class InIt, class OutIt, class Pred, class T>
    OutIt replace_copy_if(InIt first, InIt last, OutIt x,
        Pred pr, const T& val);
```

The template function executes the statement:

```
if (pr(*(first + N)))
    *(x + N) = val;
else
    *(x + N) = *(first + N)
```

once for each N in the range [0, last - first).

If x and first designate regions of storage, the range [x, x + (last - first)) must not overlap the range [first, last).

replace_if

```
template<class FwdIt, class Pred, class T>
    void replace_if(FwdIt first, FwdIt last,
        Pred pr, const T& val);
```

The template function executes the statement:

```
if (pr(*(first + N)))
    *(first + N) = val;
```

once for each N in the range [0, last - first).

reverse

```
template<class BidIt>
    void reverse(BidIt first, BidIt last);
```

The template function evaluates swap(*(first + N), *(last - 1 - N) once for each N in the range [0, (last - first) / 2). Thus, the function reverses the order of elements in the sequence.

reverse_copy

```
template<class BidIt, class OutIt>
    OutIt reverse_copy(BidIt first, BidIt last, OutIt x);
```

The template function evaluates *(x + N) = *(last - 1 - N) once for each N in the range [0, last - first). It then returns x + (last - first). Thus, the function reverses the order of elements in the sequence that it copies.

If x and first designate regions of storage, the range [x, x + (last - first)) must not overlap the range [first, last).

rotate

```
template<class FwdIt>
    void rotate(FwdIt first, FwdIt middle, FwdIt last);
```

The template function leaves the value originally stored in *(first + (N + (middle - last)) % (last - first)) subsequently stored in *(first + N) for each N in the range [0, last - first). Thus, if a ``left'' shift by one element leaves the element originally stored in *(first + (N + 1) % (last - first)) subsequently stored in *(first + N), then the function can be said to rotate the sequence either left by middle - first elements or right by last - middle elements. Both [first, middle) and [middle, last) must be valid ranges. The function swaps at most last - first pairs of elements.

rotate_copy

```
template<class FwdIt, class OutIt>
    OutIt rotate_copy(FwdIt first, FwdIt middle,
        FwdIt last, OutIt x);
```

The template function evaluates *(x + N) = *(first + (N + (middle - first)) % (last - first)) once for each N in the range [0, last - first). Thus, if a ``left'' shift by one element leaves the element originally stored in *(first + (N + 1) % (last - first)) subsequently stored in *(first + N), then the function can be said to rotate the sequence either left by middle - first elements or right by last - middle elements as it copies. Both [first, middle) and [middle, last) must be valid ranges.

If x and first designate regions of storage, the range [x, x + (last - first)) must not overlap the range [first, last).

search

```
template<class FwdIt1, class FwdIt2>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
```

```
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
```

The first template function determines the lowest value of N in the range [0, (last1 - first1) - (last2 - first2)) such that for each M in the range [0, last2 - first2), the predicate *(first1 + N + M) == *(first2 + M) is true. Here, operator== must impose an equivalence relationship between its operands. It then returns first1 + N. If no such value exists, the function returns last1. It evaluates the predicate at most (last2 - first2) * (last1 - first1) times.

The second template function behaves the same, except that the predicate is pr(*(first1 + N + M), *(first2 + M)).

search_n

```
template<class FwdIt, class Dist, class T>
    FwdIt search_n(FwdIt first, FwdIt last,
        Dist n, const T& val);
template<class FwdIt, class Dist, class T, class Pred>
    FwdIt search_n(FwdIt first, FwdIt last,
        Dist n, const T& val, Pred pr);
```

The first template function determines the lowest value of N in the range [0, (last - first) - n) such that for each M in the range [0, n), the predicate *(first + N + M) == val is true. Here, operator== must impose an equivalence relationship between its operands. It then returns first + N. If no such value exists, the function returns last. It evaluates the predicate at most n * (last - first) times.

The second template function behaves the same, except that the predicate is pr(*(first + N + M), val).

set_difference

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges [first1, last1) and [first2, last2), both ordered by operator<, to form a merged sequence of length K beginning at x, also ordered by operator<. The function then returns x + K.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering that would otherwise be copied to adjacent elements, the function copies only the element from the ordered range [first1, last1) and skips the other. An element from one sequence that has equivalent ordering with no element from the other sequence is copied from the ordered range [first1, last1) and skipped from the other. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the difference of two sets.

If x and first1 designate regions of storage, the range [x, x + K) must not overlap the range [first1, last1). If x and first2 designate regions of storage, the range [x, x + K) must not overlap the range [first2, last2). The function evaluates the ordering predicate X < Y at most 2 * ((last1 - first1) + (last2 - first2)) - 1 times.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

set_intersection

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges [first1, last1) and [first2, last2), both ordered by operator<, to form a merged sequence of length K beginning at x, also ordered by operator<. The function then returns x + K.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering that would otherwise be copied to adjacent elements, the function copies only the element from the ordered range [first1, last1) and skips the other. An element from one sequence that has equivalent ordering with no element from the other sequence is also skipped. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the intersection of two sets.

If x and first1 designate regions of storage, the range [x, x + K) must not overlap the range [first1, last1). If x and first2 designate regions of storage, the range [x, x + K) must not overlap the range [first2, last2). The function evaluates the ordering predicate X < Y at most 2 * ((last1 - first1) + (last2 - first2)) - 1 times.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

set_symmetric_difference

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_symmetric_difference(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_symmetric_difference(InIt1 first1,
        InIt1 last1, InIt2 first2, InIt2 last2, OutIt x,
            Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges [first1, last1) and [first2, last2), both ordered by operator<, to form a merged sequence of length K beginning at x, also ordered by operator<. The function then returns x + K.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering that would otherwise be copied to adjacent elements, the function copies neither element. An element from one sequence that has equivalent ordering with no element from the other sequence is copied. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the symmetric difference of two sets.

If `x` and `first1` designate regions of storage, the range `[x, x + K)` must not overlap the range `[first1, last1)`. If `x` and `first2` designate regions of storage, the range `[x, x + K)` must not overlap the range `[first2, last2)`. The function evaluates the ordering predicate `X < Y` at most `2 * ((last1 - first1) + (last2 - first2)) - 1` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

set_union

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt,
    class Pred>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges `[first1, last1)` and `[first2, last2)`, both ordered by `operator<`, to form a merged sequence of length `K` beginning at `x`, also ordered by `operator<`. The function then returns `x + K`.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering that would otherwise be copied to adjacent elements, the function copies only the element from the ordered range `[first1, last1)` and skips the other. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the union of two sets.

If `x` and `first1` designate regions of storage, the range `[x, x + K)` must not overlap the range `[first1, last1)`. If `x` and `first2` designate regions of storage, the range `[x, x + K)` must not overlap the range `[first2, last2)`. The function evaluates the ordering predicate `X < Y` at most `2 * ((last1 - first1) + (last2 - first2)) - 1` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

sort

```
template<class RanIt>
    void sort(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a sequence ordered by `operator<`. Thus, the elements are sorted in *ascending* order.

The function evaluates the ordering predicate `X < Y` at most `ceil((last - first) * log(last - first))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

sort_heap

```
template<class RanIt>
    void sort_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range [first, last) to form a sequence that is ordered by operator<. The original sequence must designate a heap, also ordered by operator<. Thus, the elements are sorted in *ascending* order.

The function evaluates the ordering predicate X < Y at most ceil((last - first) * log(last - first)) times.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

stable_partition

```
template<class BidIt, class Pred>
    BidIt stable_partition(BidIt first, BidIt last,
        Pred pr);
```

The template function reorders the sequence designated by iterators in the range [first, last) and determines the value K such that for each N in the range [0, K) the predicate pr(*(first + N)) is true, and for each N in the range [K, last - first) the predicate pr(*(first + N)) is false. It does so without altering the relative order of either the elements designated by indexes in the range [0, K) or the elements designated by indexes in the range [K, last - first). The function then returns first + K.

The predicate must not alter its operand. The function evaluates pr(*(first + N)) exactly last - first times, and swaps at most ceil((last - first) * log(last - first)) pairs of elements. (Given enough temporary storage, it can replace the swaps with at most 2 * (last - first) assignments.)

stable_sort

```
template<class BidIt>
    void stable_sort(BidIt first, BidIt last);
template<class BidIt, class Pred>
    void stable_sort(BidIt first, BidIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range [first, last) to form a sequence ordered by operator<. It does so without altering the relative order of elements that have equivalent ordering. Thus, the elements are sorted in *ascending* order.

The function evaluates the ordering predicate X < Y at most ceil((last - first) * (log(last - first))^2) times. (Given enough temporary storage, it can evaluate the predicate at most ceil((last - first) * log(last - first)) times.)

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

swap

```
template<class T>
    void swap(T& x, T& y);
```

The template function leaves the value originally stored in y subsequently stored in x, and the value originally stored in x subsequently stored in y.

swap_ranges

```
template<class FwdIt1, class FwdIt2>
    FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last,
        FwdIt2 x);
```

The template function evaluates `swap(*(first + N), *(x + N))` once for each N in the range `[0, last - first)`. It then returns `x + (last - first)`. If x and first designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

transform

```
template<class InIt, class OutIt, class Unop>
    OutIt transform(InIt first, InIt last, OutIt x,
        Unop uop);
template<class InIt1, class InIt2, class OutIt,
    class Binop>
    OutIt transform(InIt1 first1, InIt1 last1,
        InIt2 first2, OutIt x, Binop bop);
```

The first template function evaluates `*(x + N) = uop(*(first + N))` once for each N in the range `[0, last - first)`. It then returns `x + (last - first)`. The call `uop(*(first + N))` must not alter `*(first + N)`.

The second template function evaluates `*(x + N) = bop(*(first1 + N), *(first2 + N))` once for each N in the range `[0, last1 - first1)`. It then returns `x + (last1 - first1)`. The call `bop(*(first1 + N), *(first2 + N))` must not alter either `*(first1 + N)` or `*(first2 + N)`.

unique

```
template<class FwdIt>
    FwdIt unique(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt unique(FwdIt first, FwdIt last, Pred pr);
```

The first template function effectively assigns first to X, then executes the statement:

```
if (N == 0 || !(*(first + N) == V))
    V = *(first + N), *X++ = V;
```

once for each N in the range `[0, last - first)`. It then returns X. Thus, the function repeatedly removes from the sequence the second of a pair of elements for which the predicate `*(first + N) == *(first + N - 1)` is true, until only the first of a sequence of equal elements survives. Here, `operator==` must impose an equivalence relationship between its operands. It does so without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence. The function evaluates the predicate at most `last - first` times.

The second template function behaves the same, except that it executes the statement:

```
if (N == 0 || !pr(*(first + N), V))
    V = *(first + N), *X++ = V;
```

unique_copy

```
template<class InIt, class OutIt>
    OutIt unique_copy(InIt first, InIt last, OutIt x);
template<class InIt, class OutIt, class Pred>
    OutIt unique_copy(InIt first, InIt last, OutIt x,
        Pred pr);
```

The first template function effectively executes the statement:

```
if (N == 0 || !(*(first + N) == V))
    V = *(first + N), *x++ = V;
```

once for each `N` in the range `[0, last - first)`. It then returns `x`. Thus, the function repeatedly removes from the sequence it copies the second of a pair of elements for which the predicate `*(first + N) == *(first + N - 1)` is true, until only the first of a sequence of equal elements survives. Here, `operator==` must impose an equivalence relationship between its operands. It does so without altering the relative order of remaining elements, and returns the iterator value that designates the end of the copied sequence.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

The second template function behaves the same, except that it executes the statement:

```
if (N == 0 || !pr(*(first + N), V))
    V = *(first + N), *x++ = V;
```

upper_bound

```
template<class FwdIt, class T>
    FwdIt upper_bound(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt upper_bound(FwdIt first, FwdIt last,
        const T& val, Pred pr);
```

The first template function determines the highest value of `N` in the range `(0, last - first]` such that, for each `M` in the range `[0, N)` the predicate `!(val < *(first + M))` is true, where the elements designated by iterators in the range `[first, last)` form a sequence ordered by `operator<`. It then returns `first + N`. Thus, the function determines the highest position before which `val` can be inserted in the sequence and still preserve its ordering.

If `FwdIt` is a random-access iterator type, the function evaluates the ordering predicate `X < Y` at most `ceil(log(last - first)) + 1` times. Otherwise, the function evaluates the predicate a number of times proportional to `last - first`.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

<array>

Description

Include the TR1 header **<array>** to define the container template class `array` and several supporting templates.

Note: To enable this header file, you must define the macro __IBMCPP_TR1__ and use the **TARGET** compiler option to specify a valid release level. Valid release levels for TR1 support are zOSV1R12 or later. For example, you can specify the **TARGET** option as follows: `TARGET(zOSV1R12)`.

Any release prior to zOSV1R12 is invalid for the use of TR1. If TR1 code is used in an application to be compiled on an earlier platform, the compiler will issue the `#error` directive.

Synopsis

```
namespace std {
    namespace tr1 {

template<class Ty, std::size_t N>
    class array;

// TEMPLATE FUNCTIONS
template<class Ty, std::size_t N>
    bool operator==(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
template<class Ty, std::size_t N>
    bool operator!=(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
template<class Ty, std::size_t N>
    bool operator<(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
template<class Ty, std::size_t N>
    bool operator<=(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
template<class Ty, std::size_t N>
    bool operator>(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
template<class Ty, std::size_t N>
    bool operator>=(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
template<class Ty, std::size_t N>
    void swap(
        array<Ty, N>& left,
        array<Ty, N>& right);

// tuple-LIKE INTERFACE
template<int Idx, class T, std::size_t N>
     RI get(array<T, N>&);
template<int Idx, class T, std::size_t N>
     const RI get(const array<T, N>&);
template<int Idx, class T, std::size_t N>
    class tuple_element<Idx, array<T, N> >;
template<class T, std::size_t N>
    class tuple_size<array<T, N> >;
    } // namespace tr1
}  // namespace std
```

Classes

array

Description: The template class describes an object that controls a sequence of length `N` of elements of type `Ty`. The sequence is stored as an array of `Ty`, contained in the `array<Ty, N>` object.

The type has a default constructor **array()** and a default assignment operator **operator=**, and satisfies the requirements for an `aggregate`. Thus, objects of type `array<Ty, N>` can be initialized with an aggregate initializer. For example:

```
array<int, 4> ai = { 1, 2, 3 };
```

creates the object `ai` which holds four integer values, initializes the first three elements to the values 1, 2, and 3 respectively, and initializes the fourth element to 0.

Synopsis:

```
template<class Ty, std::size_t N>
    class array {
public:
    // NESTED TYPES
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;
    typedef Ty& reference;
    typedef const Ty& const_reference;
    typedef Ty *pointer;
    typedef const Ty *const_pointer;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef Ty value_type;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // CONSTRUCTORS (exposition only)
    array();
    array(const array& right);

    // MODIFICATION
    void assign(const Ty& val);
    array& operator=(const array& right);    // exposition only
    void swap(array& right);

    // ITERATORS
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;

    // SIZE QUERIES
    size_type size() const;
    size_type max_size() const;
    bool empty() const;

    // ELEMENT ACCESS
    reference at(size_type off);
    const_reference at(size_type off) const;
    reference operator[](size_type off);
    const_reference operator[](size_type off) const;
```

```
reference front();
const_reference front() const;
reference back();
const_reference back() const;

Ty *data();
const Ty *data() const;
};
```

Constructor:

array::array:

```
array();
array(const array& right);
```

The first constructor leaves the controlled sequence uninitialized (or default initialized). The second constructor initializes the controlled sequence with the sequence [right.begin(), right.end()).

Types:

array::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can server as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-specific type `T1`.

array::const_pointer:

```
typedef const Ty *const_pointer;
```

The type describes an object that can serve as a constant pointer to elements of the sequence.

array::const_reference:

```
typedef const Ty& const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

array::const_reverse_iterator:

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

The type describes an object that can server as a constant reverse iterator for the controlled sequence.

array::difference_type:

```
typedef std::ptrdiff_t difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is a synonym for the type `std::ptrdiff_t`.

array::iterator:

```
typedef T0 iterator;
```

The type describes an object that can server as a random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-specific type `T0`.

array::pointer:

```
typedef Ty *pointer;
```

The type describes an object that can serve as a pointer to elements of the sequence.

array::reference:

```
typedef Ty& reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

array::reverse_iterator:

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

The type describes an object that can server as a reverse iterator for the controlled sequence.

array::size_type:

```
typedef std::size_t size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is a synonym for the type `std::size_t`.

array::value_type:

```
typedef Ty value_type;
```

The type is a synonym for the template parameter `Ty`.

Member functions:

array::assign:

```
void assign(const Ty& val);
```

The member function replaces the sequence controlled by `*this` with a repetition of `N` elements of value `val`.

array::at:

```
reference at(size_type off);
const_reference at(size_type off) const;
```

The member functions return a reference to the element of the controlled sequence at position `off`. If that position is invalid, the function throws an object of class `out_of_range`.

array::back:

```
reference back();
const_reference back() const;
```

The member functions return a reference to the last element of the controlled sequence, which must be non-empty.

array::begin:

```
iterator begin();
const_iterator begin() const;
```

The member functions return a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

array::data:

```
Ty *data();
const Ty *data() const;
```

The member functions return the address of the first element in the controlled sequence.

array::empty:

```
bool empty() const;
```

The member function returns true only if `N == 0`.

array::end:

```
reference end();
const_reference end() const;
```

The member functions return a random-access iterator that points just beyond the end of the sequence.

array::front:

```
reference front();
const_reference front() const;
```

The member functions return a reference to the first element of the controlled sequence, which must be non-empty.

array::max_size:

```
size_type max_size() const;
```

The member function returns `N`.

array::operator[]:

```
reference operator[](size_type off);
const_reference operator[](size_type off) const;
```

The member functions return a reference to the element of the controlled sequence at position `off`. If that position is invalid, the behavior is undefined.

array::rbegin:

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The member functions return a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

array::rend:

```
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The member functions return a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence)). Hence, it designates the end of the reverse sequence.

array::size:

```
size_type size() const;
```

The member function returns `N`.

array::swap:

```
void swap(array& right);
```

The member function swaps the controlled sequences between `*this` and `right`. It performs a number of element assignments and constructor calls proportional to `N`.

Operators:

array::operator=:

```
array& operator=(const array& right);
```

The operator assigns each element of right to the corresponding element of the controlled sequence. It returns *this.

Template functions

get

```
template<int Idx, class T, std::size_t N>
    T& get(array<T, N>& arr);
template<int Idx, class T, std::size_t N>
    const T& get(const array<T, N>& arr);
```

The template functions return a reference to `arr[Idx]`.

operator!=

```
template<Ty, std::size_t N>
    bool operator!=(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
```

The template function returns `!(left == right)`.

operator==

```
template<Ty, std::size_t N>
    bool operator==(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
```

The template function overloads `operator==` to compare two objects of template class “array” on page 35. The function returns `equal(left.begin(), left.end(), right.begin())`.

operator<

```
template<Ty, std::size_t N>
    bool operator<(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
```

The template function overloads `operator<` to compare two objects of template class "array" on page 35. The function returns `lexicographical_compare(left.begin(), left.end(), right.begin())`.

operator<=

```
template<Ty, std::size_t N>
    bool operator<=(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
```

The template function returns `!(right < left)`.

operator>

```
template<Ty, std::size_t N>
    bool operator>(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
```

The template function returns `right < left`.

operator>=

```
template<Ty, std::size_t N>
    bool operator>=(
        const array<Ty, N>& left,
        const array<Ty, N>& right);
```

The template function returns `!(left < right)`.

swap

```
template<class Ty, std::size_t N>
    void swap(
        array<Ty, N>& left,
        array<Ty, N>& right);
```

The template function executes `left.swap(right)`.

Templates

tuple_element

```
template<int Idx, class T, std::size_t N>
class tuple_element<Idx, array<T, N> > {
    typedef T type;
    };
```

The template is a specialization of the template class "tuple_element" on page 399. It has a nested typedef `type` that is a synonym for the type of the `Idx` element of the `array`.

tuple_size

```
template<class T, std::size_t N>
class tuple_size<array<T, N> > {
    static const unsigned value = N;
    };
```

The template is a specialization of the template class "tuple_size" on page 399. It has a member `value` that is an integral constant expression whose value is `N`, the size of the array.

<bitset>

Description

Include the standard header **<bitset>** to define the template class `bitset` and two supporting templates.

Synopsis

```
namespace std {
template<size_t N>
    class bitset;

        // TEMPLATE FUNCTIONS
template<class E, class T, size_t N>
    basic_istream<E, T>&
        operator>>(basic_istream<E, >& is,
            bitset<N>& x);
template<class E, class T, size_t N>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const bitset<N>& x);
    }
```

Classes

bitset

Description: The template class describes an object that stores a sequence of `N` bits. A bit is **set** if its value is 1, **reset** if its value is 0. To **flip** a bit is to change its value from 1 to 0 or from 0 to 1. When converting between an object of class `bitset<N>` and an object of some integral type, bit position `j` corresponds to the bit value 1 << `j`. The integral value corresponding to two or more bits is the sum of their bit values.

Synopsis:

```
template<size_t N>
    class bitset {
public:
    typedef bool element_type;
    class reference;
    bitset();
    bitset(unsigned long val);
    template<class E, class T, class A>
        explicit bitset(const basic_string<E, T, A>& str,
            typename basic_string<E, T, A>::size_type
                pos = 0,
            typename basic_string<E, T, A>::size_type
                n = basic_string<E, T, A>::npos);
    bitset<N>& operator&=(const bitset<N>& rhs);
    bitset<N>& operator|=(const bitset<N>& rhs);
    bitset<N>& operator^=(const bitset<N>& rhs);
    bitset<N>& operator<<=(const bitset<N>& pos);
    bitset<N>& operator>>=(const bitset<N>& pos);
    bitset<N>& set();
    bitset<N>& set(size_t pos, bool val = true);
    bitset<N>& reset();
    bitset<N>& reset(size_t pos);
    bitset<N>& flip();
    bitset<N>& flip(size_t pos);
    reference operator[](size_t pos);
    bool operator[](size_t pos) const;
    reference at(size_t pos);
```

```
    bool at(size_t pos) const;
    unsigned long to_ulong() const;
    template<class E, class T, class A>
        basic_string<E, T, A> to_string() const;
    size_t count() const;
    size_t size() const;
    bool operator==(const bitset<N>& rhs) const;
    bool operator!=(const bitset<N>& rhs) const;
    bool test(size_t pos) const;
    bool any() const;
    bool none() const;
    bitset<N> operator<<(size_t pos) const;
    bitset<N> operator>>(size_t pos) const;
    bitset<N> operator~();
    static const size_t bitset_size = N;
    };
```

Constructor:

bitset::bitset:

```
bitset();
bitset(unsigned long val);
template<class E, class T, class A>
    explicit bitset(const basic_string<E, T, A>& str,
        typename basic_string<E, T, A>::size_type
            pos = 0,
        typename basic_string<E, T, A>::size_type
            n = basic_string<E, T, A>::npos);
```

The first constructor resets all bits in the bit sequence. The second constructor sets only those bits at position `j` for which `val & 1 << j` is nonzero.

The third constructor determines the initial bit values from elements of a string determined from `str`. If `str.size() < pos`, the constructor throws an object of class out_of_range. Otherwise, the effective length of the string `rlen` is the smaller of `n` and `str.size() - pos`. If any of the `rlen` elements beginning at position `pos` is other than `0` or `1`, the constructor throws an object of class invalid_argument. Otherwise, the constructor sets only those bits at position `j` for which the element at position `pos + j` is `1`.

Member functions:

bitset::any:

```
bool any() const;
```

The member function returns true if any bit is set in the bit sequence.

bitset::at:

```
bool at(size_type pos) const;
reference at(size_type pos);
```

The member function returns an object of class reference, which designates the bit at position `pos`, if the object can be modified. Otherwise, it returns the value of the bit at position `pos` in the bit sequence. If that position is invalid, the function throws an object of class out_of_range.

bitset::bitset_size:

```
static const size_t bitset_size = N;
```

The const static member is initialized to the template parameter `N`.

bitset::count:

```
size_t count() const;
```

The member function returns the number of bits set in the bit sequence.

bitset::element_type:

```
typedef bool element_type;
```

The type is a synonym for `bool`.

bitset::flip:

```
bitset<N>& flip();
bitset<N>& flip(size_t pos);
```

The first member function flips all bits in the bit sequence, then returns `*this`. The second member function throws out_of_range if `size() <= pos`. Otherwise, it flips the bit at position `pos`, then returns `*this`.

bitset::none:

```
bool none() const;
```

The member function returns true if none of the bits are set in the bit sequence.

bitset::operator!=:

```
bool operator !=(const bitset<N>& rhs) const;
```

The member operator function returns true only if the bit sequence stored in `*this` differs from the one stored in `rhs`.

bitset::operator&=:

```
bitset<N>& operator&=(const bitset<N>& rhs);
```

The member operator function replaces each element of the bit sequence stored in `*this` with the logical AND of its previous value and the corresponding bit in `rhs`. The function returns `*this`.

bitset::operator<<:

```
bitset<N> operator<<(const bitset<N>& pos);
```

The member operator function returns `bitset(*this) <<= pos`.

bitset::operator<<=:

```
bitset<N>& operator<<=(const bitset<N>& pos);
```

The member operator function replaces each element of the bit sequence stored in `*this` with the element `pos` positions earlier in the sequence. If no such earlier element exists, the function clears the bit. The function returns `*this`.

bitset::operator==:

```
bool operator ==(const bitset<N>& rhs) const;
```

The member operator function returns true only if the bit sequence stored in `*this` is the same as the one stored in `rhs`.

bitset::operator>>:

```
bitset<N> operator>>(const bitset<N>& pos);
```

The member operator function returns `bitset(*this) >>= pos`.

bitset::operator>>=:

```
bitset<N>& operator>>=(const bitset<N>& pos);
```

The member function replaces each element of the bit sequence stored in `*this` with the element `pos` positions later in the sequence. If no such later element exists, the function clears the bit. The function returns `*this`.

bitset::operator[]:

```
bool operator[](size_type pos) const;
reference operator[](size_type pos);
```

The member function returns an object of class `reference`, which designates the bit at position `pos`, if the object can be modified. Otherwise, it returns the value of the bit at position `pos` in the bit sequence. If that position is invalid, the behavior is undefined.

bitset::operator^=:

```
bitset<N>& operator^=(const bitset<N>& rhs);
```

The member operator function replaces each element of the bit sequence stored in `*this` with the logical EXCLUSIVE OR of its previous value and the corresponding bit in `rhs`. The function returns `*this`.

bitset::operator|=:

```
bitset<N>& operator|=(const bitset<N>& rhs);
```

The member operator function replaces each element of the bit sequence stored in `*this` with the logical OR of its previous value and the corresponding bit in `rhs`. The function returns `*this`.

bitset::operator~:

```
bitset<N> operator~();
```

The member operator function returns `bitset(*this).flip()`.

bitset::reference:

```
class reference {
public:
    reference& operator=(bool b};
    reference& operator=(const reference& x);
    bool operator~() const;
    operator bool() const;
    reference& flip();
    };
```

The member class describes an object that designates an individual bit within the bit sequence. Thus, for `b` an object of type `bool`, `x` and `y` objects of type `bitset<N>`, and `i` and `j` valid positions within such an object, the member functions of class `reference` ensure that (in order):

- **x[i] = b** stores `b` at bit position `i` in `x`
- **x[i] = y[j]** stores the value of the bit `y[j]` at bit position `i` in `x`
- **b = ~x[i]** stores the flipped value of the bit `x[i]` in `b`

- **b = x[i]** stores the value of the bit x[i] in b
- **x[i].flip()** stores the flipped value of the bit x[i] back at bit position i in x

bitset::reset:

```
bitset<N>& reset();
bitset<N>& reset(size_t pos);
```

The first member function resets (or clears) all bits in the bit sequence, then returns *this. The second member function throws out_of_range if size() <= pos. Otherwise, it resets the bit at position pos, then returns *this.

bitset::set:

```
bitset<N>& set();
bitset<N>& set(size_t pos, bool val = true);
```

The first member function sets all bits in the bit sequence, then returns *this. The second member function throws out_of_range if size() <= pos. Otherwise, it stores val in the bit at position pos, then returns *this.

bitset::size:

```
size_t size() const;
```

The member function returns N.

bitset::test:

```
bool test(size_t pos, bool val = true);
```

The member function throws out_of_range if size() <= pos. Otherwise, it returns true only if the bit at position pos is set.

bitset::to_string:

```
template<class E, class T, class A>
    basic_string<E, T, A> to_string() const;
```

The member function constructs str, an object of class basic_string<E, T, A>. For each bit in the bit sequence, the function appends 1 if the bit is set, otherwise 0. The *last* element appended to str corresponds to bit position zero. The function returns str.

bitset::to_ulong:

```
unsigned long to_ulong() const;
```

The member function throws overflow_error if any bit in the bit sequence has a bit value that cannot be represented as a value of type *unsigned long*. Otherwise, it returns the sum of the bit values in the bit sequence.

Template functions

operator<<

```
template<class E, class T, size_t N>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const bitset<N>& x);
```

The template function overloads `operator<<` to insert a text representation of the bit sequence in `os`. It effectively executes `os << x.to_string<E, T, allocator<E> >()`, then returns `os`.

operator>>

```
template<class E, class T, size_t N>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is,
            bitset<N>& x);
```

The template function overloads `operator>>` to store in `x` the value `bitset(str)`, where `str` is an object of type `basic_string<E, T, allocator<E> >&` extracted from `is`. The function extracts elements and appends them to `str` until:

- `N` elements have been extracted and stored
- end-of-file occurs on the input sequence
- the next input element is neither `0` not `1`, in which case the input element is not extracted

If the function stores no characters in `str`, it calls `is.setstate(ios_base::failbit)`. In any case, it returns `is`.

<cassert>

Include the standard header `<cassert>` to effectively include the standard header <assert.h>.

```
#include <assert.h>
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<cctype>

Include the standard header **<cctype>** to effectively include the standard header <ctype.h> within the std namespace.

```
#include <ctype.h>

namespace std {
using ::isalnum; using ::isalpha; using ::iscntrl;
using ::isdigit; using ::isgraph; using ::islower;
using ::isprint; using ::ispunct; using ::isspace;
using ::isupper; using ::isxdigit; using ::tolower;
using ::toupper;
}
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<cerrno>

Include the standard header `<cerrno>` to effectively include the standard header <errno.h>.

```
#include <errno.h>
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<cfloat>

Include the standard header `<cfloat>` to effectively include the standard header <float.h>.

```
#include <float.h>
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<ciso646>

Include the standard header **<ciso646>** to effectively include the standard header <iso646.h>.

```
#include <iso646.h>
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<climits>

Include the standard header **<climits>** to effectively include the standard header <limits.h>.

```
#include <limits.h>
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<clocale>

Include the standard header **<clocale>** to effectively include the standard header <locale.h> within the std namespace.

```
#include <locale.h>

namespace std {
using ::lconv; using ::localeconv; using ::setlocale;
    }
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<cmath>

Include the standard header **<cmath>** to effectively include the standard header <math.h> within the std namespace.

```
#include <math.h>

namespace std {
using ::abs; using ::acos; using ::asin;
```

```
using ::atan; using ::atan2; using ::ceil;
using ::cos; using ::cosh; using ::exp;
using ::fabs; using ::floor; using ::fmod;
using ::frexp; using ::ldexp; using ::log;
using ::log10; using ::modf; using ::pow;
using ::sin; using ::sinh; using ::sqrt;
using ::tan; using ::tanh;

using ::acosf; using ::asinf;
using ::atanf; using ::atan2f; using ::ceilf;
using ::cosf; using ::coshf; using ::expf;
using ::fabsf; using ::floorf; using ::fmodf;
using ::frexpf; using ::ldexpf; using ::logf;
using ::log10f; using ::modff; using ::powf;
using ::sinf; using ::sinhf; using ::sqrtf;
using ::tanf; using ::tanhf;

using ::acosl; using ::asinl;
using ::atanl; using ::atan2l; using ::ceill;
using ::cosl; using ::coshl; using ::expl;
using ::fabsl; using ::floorl; using ::fmodl;
using ::frexpl; using ::ldexpl; using ::logl;
using ::log10l; using ::modfl; using ::powl;
using ::sinl; using ::sinhl; using ::sqrtl;
using ::tanl; using ::tanhl;
}
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<complex>

Description

Include the standard header **<complex>** to define template class `complex` and a host of supporting template functions. Unless otherwise specified, functions that can return multiple values return an imaginary part in the half-open interval `(-pi, pi]`.

Synopsis

```
namespace std {
#define __STD_COMPLEX

        // TEMPLATE CLASSES
template<class T>
    class complex;
template<>
    class complex<float>;
template<>
    class complex<double>;
template<>
    class complex<long double>;

        // TEMPLATE FUNCTIONS
template<class T>
    complex<T> operator+(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator+(const complex<T>& lhs,
        const T& rhs);
template<class T>
    complex<T> operator+(const T& lhs,
```

```
        const complex<T>& rhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs,
        const T& rhs);
template<class T>
    complex<T> operator-(const T& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator*(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator*(const complex<T>& lhs,
        const T& rhs);
template<class T>
    complex<T> operator*(const T& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator/(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator/(const complex<T>& lhs,
        const T& rhs);
template<class T>
    complex<T> operator/(const T& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator+(const complex<T>& lhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs);
template<class T>
    bool operator==(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    bool operator==(const complex<T>& lhs,
        const T& rhs);
template<class T>
    bool operator==(const T& lhs,
        const complex<T>& rhs);
template<class T>
    bool operator!=(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    bool operator!=(const complex<T>& lhs,
        const T& rhs);
template<class T>
    bool operator!=(const T& lhs,
        const complex<T>& rhs);
template<class U, class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is,
            complex<U>& x);
template<class U, class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const complex<U>& x);
template<class T>
    T real(const complex<T>& x);
template<class T>
    T imag(const complex<T>& x);
template<class T>
    T abs(const complex<T>& x);
template<class T>
    T arg(const complex<T>& x);
template<class T>
```

```
    T norm(const complex<T>& x);
template<class T>
    complex<T> conj(const complex<T>& x);
template<class T>
    complex<T> polar(const T& rho, const T& theta = 0);
template<class T>
    complex<T> cos(const complex<T>& x);
template<class T>
    complex<T> cosh(const complex<T>& x);
template<class T>
    complex<T> exp(const complex<T>& x);
template<class T>
    complex<T> log(const complex<T>& x);
template<class T>
    complex<T> log10(const complex<T>& x);
template<class T>
    complex<T> pow(const complex<T>& x, int y);
template<class T>
    complex<T> pow(const complex<T>& x, const T& y);
template<class T>
    complex<T> pow(const complex<T>& x,
        const complex<T>& y);
template<class T>
    complex<T> pow(const T& x, const complex<T>& y);
template<class T>
    complex<T> sin(const complex<T>& x);
template<class T>
    complex<T> sinh(const complex<T>& x);
template<class T>
    complex<T> sqrt(const complex<T>& x);

namespace tr1 {
template<class T>
    complex<T> acos(complex<T>& x);
template<class T>
    complex<T> asin(complex<T>& x);
template<class T>
    complex<T> atan(complex<T>& x);
template<class T>
    complex<T> acosh(complex<T>& x);
template<class T>
    complex<T> asinh(complex<T>& x);
template<class T>
    complex<T> atanh(complex<T>& x);
template<class T>
    complex<T> fabs(complex<T>& x);
} // namespace tr1
} // namespace std
```

Macros

__STD_COMPLEX

```
#define __STD_COMPLEX
```

The macro is defined, with an unspecified expansion, to indicate compliance with the specifications of this header.

Classes

complex

Description: The template class describes an object that stores two objects of type **T**, one that represents the real part of a complex number and one that represents the imaginary part. An object of class `T`:

- has a public default constructor, destructor, copy constructor, and assignment operator — with conventional behavior
- can be assigned integer or floating-point values, or type cast to such values — with conventional behavior
- defines the arithmetic operators and math functions, as needed, that are defined for the floating-point types — with conventional behavior

In particular, no subtle differences may exist between copy construction and default construction followed by assignment. And none of the operations on objects of class T may throw exceptions.

Explicit specializations of template class complex exist for the three floating-point types. In this implementation, a value of any other type T is type cast to *double* for actual calculations, with the *double* result assigned back to the stored object of type T.

Synopsis:

```
template<class T>
    class complex {
public:
    typedef T value_type;
    T real() const;
    T imag() const;
    complex(const T& re = 0, const T& im = 0);
    template<class U>
        complex(const complex<U>& x);
    template<class U>
        complex& operator=(const complex<U>& rhs);
    template<class U>
        complex& operator+=(const complex<U>& rhs);
    template<class U>
        complex& operator-=(const complex<U>& rhs);
    template<class U>
        complex& operator*=(const complex<U>& rhs);
    template<class U>
        complex& operator/=(const complex<U>& rhs);
    complex& operator=(const T& rhs);
    complex& operator+=(const T& rhs);
    complex& operator-=(const T& rhs);
    complex& operator*=(const T& rhs);
    complex& operator/=(const T& rhs);
    friend complex<T>
        operator+(const complex<T>& lhs, const T& rhs);
    friend complex<T>
        operator+(const T& lhs, const complex<T>& rhs);
    friend complex<T>
        operator-(const complex<T>& lhs, const T& rhs);
    friend complex<T>
        operator-(const T& lhs, const complex<T>& rhs);
    friend complex<T>
        operator*(const complex<T>& lhs, const T& rhs);
    friend complex<T>
        operator*(const T& lhs, const complex<T>& rhs);
    friend complex<T>
        operator/(const complex<T>& lhs, const T& rhs);
    friend complex<T>
        operator/(const T& lhs, const complex<T>& rhs);
    friend bool
        operator==(const complex<T>& lhs, const T& rhs);
    friend bool
        operator==(const T& lhs, const complex<T>& rhs);
    friend bool
```

```
        operator!=(const complex<T>& lhs, const T& rhs);
    friend bool
        operator!=(const T& lhs, const complex<T>& rhs);
    };
```

Constructor:

complex::complex:

```
complex(const T& re = 0, const T& im = 0);
template<class U>
    complex(const complex<U>& x);
```

The first constructor initializes the stored real part to `re` and the stored imaginary part to `im`. The second constructor initializes the stored real part to `x.real()` and the stored imaginary part to `x.imag()`.

In this implementation, if a translator does not support member template functions, the template:

```
template<class U>
    complex(const complex<U>& x);
```

is replaced by:

```
complex(const complex& x);
```

which is the copy constructor.

Member functions:

complex::imag:

```
T imag() const;
```

The member function returns the stored imaginary part.

complex::operator=:*

```
template<class U>
    complex& operator*=(const complex<U>& rhs);
complex& operator*=(const T& rhs);
```

The first member function replaces the stored real and imaginary parts with those corresponding to the complex product of `*this` and `rhs`. It then returns `*this`.

The second member function multiplies both the stored real part and the stored imaginary part with `rhs`. It then returns `*this`.

In this implementation, if a translator does not support member template functions, the template:

```
template<class U>
    complex& operator*=(const complex<U>& rhs);
```

is replaced by:

```
complex& operator*=(const complex& rhs);
```

complex::operator+=:

```
template<class U>
    complex& operator+=(const complex<U>& rhs);
complex& operator+=(const T& rhs);
```

The first member function replaces the stored real and imaginary parts with those corresponding to the complex sum of `*this` and `rhs`. It then returns `*this`.

The second member function adds `rhs` to the stored real part. It then returns `*this`.

In this implementation, if a translator does not support member template functions, the template:

```
template<class U>
    complex& operator+=(const complex<U>& rhs);
```

is replaced by:

```
complex& operator+=(const complex& rhs);
```

complex::operator-=:

```
template<class U>
    complex& operator-=(const complex<U>& rhs);
complex& operator-=(const T& rhs);
```

The first member function replaces the stored real and imaginary parts with those corresponding to the complex difference of `*this` and `rhs`. It then returns `*this`.

The second member function subtracts `rhs` from the stored real part. It then returns `*this`.

In this implementation, if a translator does not support member template functions, the template:

```
template<class U>
    complex& operator-=(const complex<U>& rhs);
```

is replaced by:

```
complex& operator-=(const complex& rhs);
```

complex::operator/=:

```
template<class U>
    complex& operator/=(const complex<U>& rhs);
complex& operator/=(const T& rhs);
```

The first member function replaces the stored real and imaginary parts with those corresponding to the complex quotient of `*this` and `rhs`. It then returns `*this`.

The second member function multiplies both the stored real part and the stored imaginary part with `rhs`. It then returns `*this`.

In this implementation, if a translator does not support member template functions, the template:

```
template<class U>
    complex& operator/=(const complex<U>& rhs);
```

is replaced by:

```
complex& operator/=(const complex& rhs);
```

complex::operator=:

```
template<class U>
    complex& operator=(const complex<U>& rhs);
complex& operator=(const T& rhs);
```

The first member function replaces the stored real part with `rhs.real()` and the stored imaginary part with `rhs.imag()`. It then returns `*this`.

The second member function replaces the stored real part with `rhs` and the stored imaginary part with zero. It then returns `*this`.

In this implementation, if a translator does not support member template functions, the template:

```
template<class U>
    complex& operator=(const complex<U>& rhs);
```

is replaced by:

```
complex& operator=(const complex& rhs);
```

which is the default assignment operator.

complex::real:

```
T real() const;
```

The member function returns the stored real part.

complex::value_type:

```
typedef T value_type;
```

The type is a synonym for the template parameter `T`.

complex<float>

```
template<>
    class complex<float> {
public:
    complex(float re = 0, float im = 0);
    explicit complex(const complex<double>& x);
    explicit complex(const complex<long double>& x);
// rest same as template class complex
    };
```

The explicitly specialized template class describes an object that stores two objects of type **float**, one that represents the real part of a complex number and one that represents the imaginary part. The explicit specialization differs only in the constructors it defines. The first constructor initializes the stored real part to `re` and the stored imaginary part to `im`. The remaining two constructors initialize the stored real part to `x.real()` and the stored imaginary part to `x.imag()`.

complex<double>

```
template<>
    class complex<double> {
public:
    complex(double re = 0, double im = 0);
    complex(const complex<float>& x);
    explicit complex(const complex<long double>& x);
// rest same as template class complex
    };
```

The explicitly specialized template class describes an object that stores two objects of type **double**, one that represents the real part of a complex number and one that represents the imaginary part. The explicit specialization differs only in the constructors it defines. The first constructor initializes the stored real part to `re` and the stored imaginary part to im. The remaining two constructors initialize the stored real part to x.real() and the stored imaginary part to x.imag().

complex<long double>

```
template<>
    class complex<long double> {
public:
    complex(long double re = 0, long double im = 0);
    complex(const complex<float>& x);
    complex(const complex<double>& x);
// rest same as template class complex
    };
```

The explicitly specialized template class describes an object that stores two objects of type **long double**, one that represents the real part of a complex number and one that represents the imaginary part. The explicit specialization differs only in the constructors it defines. The first constructor initializes the stored real part to re and the stored imaginary part to im. The remaining two constructors initialize the stored real part to x.real() and the stored imaginary part to x.imag().

Template functions

abs

```
template<class T>
    T abs(const complex<T>& x);
```

The function returns the magnitude of x.

acos

```
template<class T>
    complex<T> acos(complex<T>& x);
```

The **acos** function computes the complex arc cosine of x, with branch cuts outside the interval [-1, +1] along the real axis.

From z/OS V1R13 XL C/C++, you can use C99 library facilities in your C++ code. This new TR1 template function **acos** produces the same results as the existing C99 functions, as indicated in the following table.

TR1 template function	Corresponding C99 function
template <> complex<float> **acos**(complex<float>& x);	float complex **cacosf**(float complex z);
template <> complex<double> **acos**(complex<double>& x);	double complex **cacos**(double complex z);
template <> complex<long double> **acos**(complex<long double>& x);	long double complex **cacosl**(long double complex z);

Note: To enable the TR1 headers, you must define the macro **__IBMCPP_TR1__** as 1.

acosh

```
template<class T>
    complex<T> acosh(complex<T>& x);
```

The **acosh** function computes the complex arc hyperbolic cosine of the x parameter, with a branch cut at values less than 1 along the real axis.

From z/OS V1R13 XL C/C++, you can use C99 library facilities in your C++ code. This new TR1 template function **acosh** produces the same results as the existing C99 functions, as indicated in the following table.

TR1 template function	Corresponding C99 function
`template <> complex<float> acosh(complex<float>& x);`	`float complex cacoshf(float complex z);`
`template <> complex<double> acosh(complex<double>& x);`	`double complex cacosh(double complex z);`
`template <> complex<long double> acosh(complex<long double>& x);`	`long double complex cacoshl(long double complex z);`

Note: To enable the TR1 headers, you must define the macro **__IBMCPP_TR1__** as 1.

arg

```
template<class T>
    T arg(const complex<T>& x);
```

The function returns the phase angle of `x`.

asin

```
template<class T>
    complex<T> asin(complex<T>& x);
```

The **asin** function computes the complex arc sine `x`, with branch cuts outside the interval [-1, +1] along the real axis.

From z/OS V1R13 XL C/C++, you can use C99 library facilities in your C++ code. This new TR1 template function **asin** produces the same results as the existing C99 functions, as indicated in the following table.

TR1 template function	Corresponding C99 function
`template <> complex<float> asin(complex<float>& x);`	`float complex casinf(float complex z);`
`template <> complex<double> asin(complex<double>& x);`	`double complex casin(double complex z);`
`template <> complex<long double> asin(complex<long double>& x);`	`long double complex casinl(long double complex z);`

Note: To enable the TR1 headers, you must define the macro **__IBMCPP_TR1__** as 1.

asinh

```
template<class T>
    complex<T> asinh(complex<T>& x);
```

The **asinh** function computes the complex arc hyperbolic sine of the `x` parameter, with branch cuts outside the interval [-i, +i] along the imaginary axis.

From z/OS V1R13 XL C/C++, you can use C99 library facilities in your C++ code. This new TR1 template function **asinh** produces the same results as the existing C99 functions, as indicated in the following table.

TR1 template function	Corresponding C99 function
template <> complex<float> **asinh**(complex<float>& x);	float complex **casinhf**(float complex z);
template <> complex<double> **asinh**(complex<double>& x);	double complex **casinh**(double complex z);
template <> complex<long double> **asinh**(complex<long double>& x);	long double complex **casinhl**(long double complex z);

Note: To enable the TR1 headers, you must define the macro **__IBMCPP_TR1__** as 1.

atan

```
template<class T>
    complex<T> atan(complex<T>& x);
```

The **atan** function computes the complex arc tangent of x, with branch cuts outside the interval [-i, +i] along the imaginary axis.

From z/OS V1R13 XL C/C++, you can use C99 library facilities in your C++ code. This new TR1 template function **atan** produces the same results as the existing C99 functions, as indicated in the following table.

TR1 template function	Corresponding C99 function
template <> complex<float> **atan**(complex<float>& x);	float complex **catanf**(float complex z);
template <> complex<double> **atan**(complex<double>& x);	double complex **catan**(double complex z);
template <> complex<long double> **atan**(complex<long double>& x);	long double complex **catanl**(long double complex z);

Note: To enable the TR1 headers, you must define the macro **__IBMCPP_TR1__** as 1.

atanh

```
template<class T>
    complex<T> atanh(complex<T>& x);
```

The **atanh** function computes the complex arc hyperbolic tangent of x, with branch cuts outside the interval [-1, +1] along the real axis.

From z/OS V1R13 XL C/C++, you can use C99 library facilities in your C++ code. This new TR1 template function **atanh** produces the same results as the existing C99 functions, as indicated in the following table.

TR1 template function	Corresponding C99 function
template <> complex<float> **atanh**(complex<float>& x);	float complex **catanhf**(float complex z);
template <> complex<double> **atanh**(complex<double>& x);	double complex **catanh**(double complex z);
template <> complex<long double> **atanh**(complex<long double>& x);	long double complex **catanhl**(long double complex z);

Note: To enable the TR1 headers, you must define the macro **__IBMCPP_TR1__** as 1.

conj

```
template<class T>
    complex<T> conj(const complex<T>& x);
```

The function returns the conjugate of x.

cos

```
template<class T>
    complex<T> cos(const complex<T>& x);
```

The function returns the cosine of x.

cosh

```
template<class T>
    complex<T> cosh(const complex<T>& x);
```

The function returns the hyperbolic cosine of x.

exp

```
template<class T>
    complex<T> exp(const complex<T>& x);
```

The function returns the exponential of x.

fabs

```
template<class T>
    complex<T> fabs(complex<T>& x);
```

The **fabs** function computes the complex absolute value of x.

From z/OS V1R13 XL C/C++, you can use C99 library facilities in your C++ code. This new TR1 template function **fabs** produces the same results as the existing C99 functions, as indicated in the following table.

TR1 template function	Corresponding C99 function
template <> complex<float> **fabs**(complex<float>& x);	float complex **cabsf**(float complex z);
template <> complex<double> **fabs**(complex<double>& x);	double complex **cabs**(double complex z);
template <> complex<long double> **fabs**(complex<long double>& x);	long double complex **cabsl**(long double complex z);

Note: To enable the TR1 headers, you must define the macro **__IBMCPP_TR1__** as 1.

imag

```
template<class T>
    T imag(const complex<T>& x);
```

The function returns the imaginary part of x.

log

```
template<class T>
    complex<T> log(const complex<T>& x);
```

The function returns the logarithm of x. The branch cuts are along the negative real axis.

log10

```
template<class T>
    complex<T> log10(const complex<T>& x);
```

The function returns the base 10 logarithm of x. The branch cuts are along the negative real axis.

norm

```
template<class T>
    T norm(const complex<T>& x);
```

The function returns the squared magnitude of x.

operator!=

```
template<class T>
    bool operator!=(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    bool operator!=(const complex<T>& lhs,
        const T& rhs);
template<class T>
    bool operator!=(const T& lhs,
        const complex<T>& rhs);
```

The operators each return true only if `real(lhs) != real(rhs) || imag(lhs) != imag(rhs)`.

operator*

```
template<class T>
    complex<T> operator*(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator*(const complex<T>& lhs,
        const T& rhs);
template<class T>
    complex<T> operator*(const T& lhs,
        const complex<T>& rhs);
```

The operators each convert both operands to the return type, then return the complex product of the converted lhs and rhs.

operator+

```
template<class T>
    complex<T> operator+(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator+(const complex<T>& lhs,
        const T& rhs);
template<class T>
    complex<T> operator+(const T& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator+(const complex<T>& lhs);
```

The binary operators each convert both operands to the return type, then return the complex sum of the converted lhs and rhs.

The unary operator returns lhs.

operator-

```
template<class T>
    complex<T> operator-(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs,
        const T& rhs);
template<class T>
    complex<T> operator-(const T& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs);
```

The binary operators each convert both operands to the return type, then return the complex difference of the converted `lhs` and `rhs`.

The unary operator returns a value whose real part is `-real(lhs)` and whose imaginary part is `-imag(lhs)`.

operator/

```
template<class T>
    complex<T> operator/(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    complex<T> operator/(const complex<T>& lhs,
        const T& rhs);
template<class T>
    complex<T> operator/(const T& lhs,
        const complex<T>& rhs);
```

The operators each convert both operands to the return type, then return the complex quotient of the converted `lhs` and `rhs`.

operator<<

```
template<class U, class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const complex<U>& x);
```

The template function inserts the complex value `x` in the output stream `os`, effectively by executing:

```
basic_ostringstream<E, T> ostr;
ostr.flags(os.flags());
ostr.imbue(os.imbue());
ostr.precision(os.precision());
ostr << '(' << real(x) << ','
    << imag(x) << ')';
os << ostr.str().c_str();
```

Thus, if `os.width()` is greater than zero, any padding occurs either before or after the parenthesized pair of values, which itself contains no padding. The function returns `os`.

operator==

```
template<class T>
    bool operator==(const complex<T>& lhs,
        const complex<T>& rhs);
template<class T>
    bool operator==(const complex<T>& lhs,
```

```
        const T& rhs);
template<class T>
    bool operator==(const T& lhs,
        const complex<T>& rhs);
```

The operators each return true only if `real(lhs) == real(rhs) && imag(lhs) == imag(rhs)`.

operator>>

```
template<class U, class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is,
            complex<U>& x);
```

The template function attempts to extract a complex value from the input stream `is`, effectively by executing:

```
is >> ch && ch == '('
    && is >> re >> ch && ch == ','
    && is >> im >> ch && ch == ')'
```

Here, `ch` is an object of type `E`, and `re` and `im` are objects of type `U`.

If the result of this expression is true, the function stores `re` in the real part and `im` in the imaginary part of `x`. In any event, the function returns `is`.

polar

```
template<class T>
    complex<T> polar(const T& rho,
        const T& theta = 0);
```

The function returns the complex value whose magnitude is `rho` and whose phase angle is `theta`.

pow

```
template<class T>
    complex<T> pow(const complex<T>& x, int y);
template<class T>
    complex<T> pow(const complex<T>& x,
        const T& y);
template<class T>
    complex<T> pow(const complex<T>& x,
        const complex<T>& y);
template<class T>
    complex<T> pow(const T& x,
        const complex<T>& y);
```

The functions each effectively convert both operands to the return type, then return the converted `x` to the power `y`. The branch cut for `x` is along the negative real axis.

real

```
template<class T>
    T real(const complex<T>& x);
```

The function returns the real part of `x`.

sin

```
template<class T>
    complex<T> sin(const complex<T>& x);
```

The function returns the sine of x.

sinh

```
template<class T>
    complex<T> sinh(const complex<T>& x);
```

The function returns the hyperbolic sine of x.

sqrt

```
template<class T>
    complex<T> sqrt(const complex<T>& x);
```

The function returns the square root of x, with phase angle in the half-open interval (-pi/2, pi/2]. The branch cuts are along the negative real axis.

tan

```
template<class T>
    complex<T> tan(const complex<T>& x);
```

The function returns the tangent of x.

tanh

```
template<class T>
    complex<T> tanh(const complex<T>& x);
```

The function returns the hyperbolic tangent of x.

<csetjmp>

Include the standard header **<csetjmp>** to effectively include the standard header <setjmp.h> within the std namespace.

```
#include <setjmp.h>

namespace std {
using ::jmp_buf; using ::longjmp;
    }
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<csignal>

Include the standard header **<csignal>** to effectively include the standard header <signal.h> within the std namespace.

```
#include <signal.h>

namespace std {
using ::sig_atomic_t; using ::raise; using ::signal;
    }
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<cstdarg>

Include the standard header **<cstdarg>** to effectively include the standard header <stdarg.h> within the std namespace.

```
#include <stdarg.h>

namespace std {
using ::va_list;
    }
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<cstddef>

Include the standard header **<cstddef>** to effectively include the standard header <stddef.h> within the std namespace.

```
#include <stddef.h>

namespace std {
using ::ptrdiff_t; using ::size_t;
    }
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<cstdio>

Include the standard header **<cstdio>** to effectively include the standard header <stdio.h> within the std namespace.

```
#include <stdio.h>

namespace std {
using ::size_t; using ::fpos_t; using ::FILE;
using ::clearerr; using ::fclose; using ::feof;
using ::ferror; using ::fflush; using ::fgetc;
using ::fgetpos; using ::fgets; using ::fopen;
using ::fprintf; using ::fputc; using ::fputs;
using ::fread; using ::freopen; using ::fscanf;
using ::fseek; using ::fsetpos; using ::ftell;
using ::fwrite; using ::gets; using ::perror;
using ::printf; using ::puts; using ::remove;
using ::rename; using ::rewind; using ::scanf;
using ::setbuf; using ::setvbuf; using ::sprintf;
using ::sscanf; using ::tmpfile; using ::tmpnam;
using ::ungetc; using ::vfprintf; using ::vprintf;
using ::vsprintf;
}
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<cstdlib>

Include the standard header **<cstdlib>** to effectively include the standard header <stdlib.h> within the std namespace.

```
#include <stdlib.h>

namespace std {
using ::size_t; using ::div_t; using ::ldiv_t;
using ::abort; using ::abs; using ::atexit;
using ::atof; using ::atoi; using ::atol;
using ::bsearch; using ::calloc; using ::div;
using ::exit; using ::free; using ::getenv;
using ::labs; using ::ldiv; using ::malloc;
using ::mblen; using ::mbstowcs; using ::mbtowc;
using ::qsort; using ::rand; using ::realloc;
using ::srand; using ::strtod; using ::strtol;
using ::strtoul; using ::system;
using ::wcstombs; using ::wctomb;
}
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<cstring>

Include the standard header **<cstring>** to effectively include the standard header <string.h> within the std namespace.

```
#include <string.h>

namespace std {
using ::size_t; using ::memcmp; using ::memcpy;
using ::memmove; using ::memset; using ::strcat;
using ::strcmp; using ::strcoll; using ::strcpy;
using ::strcspn; using ::strerror; using ::strlen;
using ::strncat; using ::strncmp; using ::strncpy;
using ::strspn; using ::strtok; using ::strxfrm;
    }
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<ctime>

Include the standard header **<ctime>** to effectively include the standard header <time.h> within the std namespace.

```
#include <time.h>

namespace std {
using ::clock_t; using ::size_t;
using ::time_t; using ::tm;
using ::asctime; using ::clock; using ::ctime;
using ::difftime; using ::gmtime; using ::localtime;
using ::mktime; using ::strftime; using ::time;
    }
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<cwchar>

Include the standard header **<cwchar>** to effectively include the standard header <wchar.h> within the std namespace.

```
#include <wchar.h>

namespace std {
using ::mbstate_t; using ::size_t; using ::wint_t;
using ::fgetwc; using ::fgetws; using ::fputwc;
using ::fputws; using ::fwide; using ::fwprintf;
using ::fwscanf; using ::getwc; using ::getwchar;
using ::mbrlen; using ::mbrtowc; using ::mbsrtowcs;
using ::mbsinit; using ::putwc; using ::putwchar;
using ::swprintf; using ::swscanf; using ::ungetwc;
using ::vfwprintf; using ::vswprintf; using ::vwprintf;
using ::wcrtomb; using ::wprintf; using ::wscanf;
using ::wcsrtombs; using ::wcstol; using ::wcscat;
using ::wcschr; using ::wcscmp; using ::wcscoll;
using ::wcscpy; using ::wcscspn; using ::wcslen;
using ::wcsncat; using ::wcsncmp; using ::wcsncpy;
using ::wcspbrk; using ::wcsrchr; using ::wcsspn;
using ::wcsstr; using ::wcstok; using ::wcsxfrm;
using ::wmemchr; using ::wmemcmp; using ::wmemcpy;
using ::wmemmove; using ::wmemset; using ::wcsftime;
}
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<cwctype>

Include the standard header **<cwctype>** to effectively include the standard header <wctype.h> within the std namespace.

```
#include <wctype.h>

namespace std {
using ::wint_t; using ::wctrans_t; using ::wctype_t;
using ::iswalnum; using ::iswalpha; using ::iswcntrl;
using ::iswctype; using ::iswdigit; using ::iswgraph;
using ::iswlower; using ::iswprint; using ::iswpunct;
using ::iswspace; using ::iswupper; using ::iswxdigit;
using ::towctrans; using ::towlower; using ::towupper;
using ::wctrans; using ::wctype;
}
```

Note: The Standard C++ library works in conjunction with the headers from the Standard C library. For information about the Standard C library, refer to the documentation that is supplied with the operating system.

<deque>

Description

Include the STL standard header **<deque>** to define the container template class deque and several supporting templates.

Synopsis

```
namespace std {
template<class T, class A>
    class deque;

        // TEMPLATE FUNCTIONS
template<class T, class A>
    bool operator==(
```

```
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator!=(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator<(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator<=(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    void swap(
        deque<T, A>& lhs,
        deque<T, A>& rhs);
    }
```

Classes

deque

Description: The template class describes an object that controls a varying-length sequence of elements of type `T`. The sequence is represented in a way that permits insertion and removal of an element at either end with a single element copy (constant time). Such operations in the middle of the sequence require element copies and assignments proportional to the number of elements in the sequence (linear time).

The object allocates and frees storage for the sequence it controls through a stored allocator object of class `A`. Such an allocator object must have the same external interface as an object of template class allocator. Note that the stored allocator object is *not* copied when the container object is assigned.

Deque reallocation occurs when a member function must insert or erase elements of the controlled sequence:

- If an element is inserted into an empty sequence, or if an element is erased to leave an empty sequence, then iterators earlier returned by `begin()` and `end()` become **invalid**.
- If an element is inserted at `first()`, then all iterators but no references, that designate existing elements become invalid.
- If an element is inserted at `end()`, then `end()` and all iterators, but no references, that designate existing elements become invalid.
- If an element is erased at `first()`, only that iterator and references to the erased element become invalid.
- If an element is erased at `last() - 1`, only that iterator, `last()`, and references to the erased element become invalid.
- Otherwise, inserting or erasing an element invalidates all iterators and references.

Synopsis:

```
template<class T, class A = allocator<T> >
    class deque {
public:
    typedef A allocator_type;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::value_type value_type;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator>
        reverse_iterator;
    deque();
    explicit deque(const A& al);
    explicit deque(size_type n);
    deque(size_type n, const T& v);
    deque(size_type n, const T& v,
        const A& al);
    deque(const deque& x);
    template<class InIt>
        deque(InIt first, InIt last);
    template<class InIt>
        deque(InIt first, InIt last, const A& al);
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    void resize(size_type n);
    void resize(size_type n, T x);
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    reference at(size_type pos);
    const_reference at(size_type pos) const;
    reference operator[](size_type pos);
    const_reference operator[](size_type pos);
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
    void push_front(const T& x);
    void pop_front();
    void push_back(const T& x);
    void pop_back();
    template<class InIt>
        void assign(InIt first, InIt last);
    void assign(size_type n, const T& x);
    iterator insert(iterator it, const T& x);
    void insert(iterator it, size_type n, const T& x);
    template<class InIt>
        void insert(iterator it, InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    void clear();
    void swap(deque& x);
    };
```

Constructor:

deque::deque:

```
deque();
explicit deque(const A& al);
explicit deque(size_type n);
deque(size_type n, const T& v);
deque(size_type n, const T& v,
    const A& al);
deque(const deque& x);
template<class InIt>
    deque(InIt first, InIt last);
template<class InIt>
    deque(InIt first, InIt last, const A& al);
```

All constructors store an allocator object and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

The first two constructors specify an empty initial controlled sequence. The third constructor specifies a repetition of `n` elements of value `T()`. The fourth and fifth constructors specify a repetition of `n` elements of value `x`. The sixth constructor specifies a copy of the sequence controlled by `x`. If `InIt` is an integer type, the last two constructors specify a repetition of `(size_type)first` elements of value `(T)last`. Otherwise, the last two constructors specify the sequence `[first, last)`.

Types:

deque::allocator_type:

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

deque::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

deque::const_pointer:

```
typedef typename A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

deque::const_reference:

```
typedef typename A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

deque::const_reverse_iterator:

```
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse random-access iterator for the controlled sequence.

deque::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

deque::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T0.

deque::pointer:

```
typedef typename A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

deque::reference:

```
typedef typename A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

deque::reverse_iterator:

```
typedef reverse_iterator<iterator>
    reverse_iterator;
```

The type describes an object that can serve as a reverse random-access iterator for the controlled sequence.

deque::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

deque::value_type:

```
typedef typename A::value_type value_type;
```

The type is a synonym for the template parameter T.

Member functions:

deque::assign:

```
template<class InIt>
    void assign(InIt first, InIt last);
void assign(size_type n, const T& x);
```

If InIt is an integer type, the first member function behaves the same as assign((size_type)first, (T)last). Otherwise, the first member function replaces the sequence controlled by *this with the sequence [first, last), which must *not* overlap the initial controlled sequence. The second member function replaces the sequence controlled by *this with a repetition of n elements of value x.

deque::at:

```
const_reference at(size_type pos) const;
reference at(size_type pos);
```

The member function returns a reference to the element of the controlled sequence at position pos. If that position is invalid, the function throws an object of class out_of_range.

deque::back:

```
reference back();
const_reference back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

deque::begin:

```
const_iterator begin() const;
iterator begin();
```

The member function returns a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

deque::clear:

```
void clear();
```

The member function calls erase(begin(), end()).

deque::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

deque::end:

```
const_iterator end() const;
iterator end();
```

The member function returns a random-access iterator that points just beyond the end of the sequence.

deque::erase:

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by it. The second member function removes the elements of the controlled sequence in the range [first, last). Both return an iterator that designates the first element remaining beyond any elements removed, or end() if no such element exists.

Removing N elements causes N destructor calls and an assignment for each of the elements between the insertion point and the nearer end of the sequence. Removing an element at either end invalidates only iterators and references that designate the erased elements. Otherwise, erasing an element invalidates all iterators and references.

The member functions never throw an exception.

deque::front:

```
reference front();
const_reference front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

deque::get_allocator:

```
A get_allocator() const;
```

The member function returns the stored allocator object.

deque::insert:

```
iterator insert(iterator it, const T& x);
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
```

Each of the member functions inserts, before the element pointed to by `it` in the controlled sequence, a sequence specified by the remaining operands. The first member function inserts a single element with value `x` and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of `n` elements of value `x`.

If `InIt` is an integer type, the last member function behaves the same as `insert(it, (size_type)first, (T)last)`. Otherwise, the last member function inserts the sequence `[first, last)`, which must *not* overlap the initial controlled sequence.

When inserting a single element, the number of element copies is linear in the number of elements between the insertion point and the nearer end of the sequence. When inserting a single element at either end of the sequence, the amortized number of element copies is constant. When inserting `N` elements, the number of element copies is linear in `N` plus the number of elements between the insertion point and the nearer end of the sequence — except when the template member is specialized for `InIt` an input or forward iterator, which behaves like `N` single insertions. Inserting an element at either end invalidates all iterators, but no references, that designate existing elements. Otherwise, inserting an element invalidates all iterators and references.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, and the exception is not thrown while copying an element, the container is left unaltered and the exception is rethrown.

deque::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

deque::operator[]:

```
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
```

The member function returns a reference to the element of the controlled sequence at position pos. If that position is invalid, the behavior is undefined.

deque::pop_back:

```
void pop_back();
```

The member function removes the last element of the controlled sequence, which must be non-empty. Removing the element invalidates only iterators and references that designate the erased element.

The member function never throws an exception.

deque::pop_front:

```
void pop_front();
```

The member function removes the first element of the controlled sequence, which must be non-empty. Removing the element invalidates only iterators and references that designate the erased element.

The member function never throws an exception.

deque::push_back:

```
void push_back(const T& x);
```

The member function inserts an element with value x at the end of the controlled sequence. Inserting the element invalidates all iterators, but no references, to existing elements.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

deque::push_front:

```
void push_front(const T& x);
```

The member function inserts an element with value x at the beginning of the controlled sequence. Inserting the element invalidates all iterators, but no references, to existing elements.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

deque::rbegin:

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

deque::rend:

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

deque::resize:

```
void resize(size_type n);
void resize(size_type n, T x);
```

The member functions both ensure that `size()` henceforth returns `n`. If it must make the controlled sequence longer, the first member function appends elements with value `T()`, while the second member function appends elements with value `x`. To make the controlled sequence shorter, both member functions call `erase(begin() + n, end())`.

deque::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

deque::swap:

```
void swap(deque& x);
```

The member function swaps the controlled sequences between `*this` and `x`. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Template functions

operator!=

```
template<class T, class A>
    bool operator!=(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class A>
    bool operator==(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function overloads `operator==` to compare two objects of template class deque. The function returns `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end(), rhs.begin())`.

operator<

```
template<class T, class A>
    bool operator<(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function overloads `operator<` to compare two objects of template class deque. The function returns `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class T, class A>
    bool operator<=(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class A>
    bool operator>(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class A>
    bool operator>=(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class T, class A>
    void swap(
        deque <T, A>& lhs,
        deque <T, A>& rhs);
```

The template function executes `lhs.swap (rhs)`.

<exception>

Description

Include the standard header **<exception>** to define several types and functions related to the handling of exceptions.

Synopsis

```
namespace std {
class exception;
class bad_exception;

        // FUNCTIONS
typedef void (*terminate_handler)();
typedef void (*unexpected_handler)();
terminate_handler
    set_terminate(terminate_handler ph) throw();
```

```
unexpected_handler
    set_unexpected(unexpected_handler ph) throw();
void terminate();
void unexpected();
bool uncaught_exception();
    }
```

Classes

bad_exception

```
class bad_exception : public exception {
    };
```

The class describes an exception that can be thrown from an unexpected handler. The value returned by `what()` is an implementation-defined C string. None of the member functions throw any exceptions.

exception

```
class exception {
public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
    };
```

The class serves as the base class for all exceptions thrown by certain expressions and by the Standard C++ library. The C string value returned by **what()** is left unspecified by the default constructor, but may be defined by the constructors for certain derived classes as an implementation-defined C string.

None of the member functions throw any exceptions.

Functions

set_terminate

```
terminate_handler
    set_terminate(terminate_handler ph) throw();
```

The function establishes a new terminate handler as the function `*ph`. Thus, `ph` must not be a null pointer. The function returns the address of the previous terminate handler.

set_unexpected

```
unexpected_handler
    set_unexpected(unexpected_handler ph) throw();
```

The function establishes a new unexpected handler as the function `*ph`. Thus, `ph` must not be a null pointer. The function returns the address of the previous unexpected handler.

terminate

```
void terminate();
```

The function calls a **terminate handler**, a function of type `void ()`. If `terminate` is called directly by the program, the terminate handler is the one most recently set by a call to set_terminate. If `terminate` is called for any of several other reasons during evaluation of a throw expression, the terminate handler is the one in effect immediately after evaluating the throw expression.

A terminate handler may not return to its caller. At program startup, the terminate handler is a function that calls `abort()`.

uncaught_exception

```
bool uncaught_exception();
```

The function returns true only if a thrown exception is being currently processed. Specifically, it returns true after completing evaluation of a throw expression and before completing initialization of the exception declaration in the matching handler or calling unexpected as a result of the throw expression.

unexpected

```
void unexpected();
```

The function calls an **unexpected handler**, a function of type `void ()`. If `unexpected` is called directly by the program, the unexpected handler is the one most recently set by a call to set_unexpected. If `unexpected` is called when control leaves a function by a thrown exception of a type not permitted by an **exception specification** for the function, as in:

```
void f() throw()      // function may throw no exceptions
    {throw "bad"; }   // throw calls unexpected()
```

the unexpected handler is the one in effect immediately after evaluating the throw expression.

An unexpected handler may not return to its caller. It may terminate execution by:
- throwing an object of a type listed in the exception specification (or an object of any type if the unexpected handler is called directly by the program)
- throwing an object of type `bad_exception`
- calling `terminate()`, `abort()`, or `exit(int)`

At program startup, the unexpected handler is a function that calls `terminate()`.

Types

terminate_handler

```
typedef void (*terminate_handler)();
```

The type describes a pointer to a function suitable for use as a terminate handler.

unexpected_handler

```
typedef void (*unexpected_handler)();
```

The type describes a pointer to a function suitable for use as an unexpected handler.

<fstream>

Description

Include the iostreams standard header **<fstream>** to define several classes that support iostreams operations on sequences stored in external files.

Synopsis

```
namespace std {
template<class E, class T = char_traits<E> >
    class basic_filebuf;
typedef basic_filebuf<char> filebuf;
typedef basic_filebuf<wchar_t> wfilebuf;
template<class E, class T = char_traits<E> >
    class basic_ifstream;
typedef basic_ifstream<char> ifstream;
typedef basic_ifstream<wchar_t> wifstream;
template<class E, class T = char_traits<E> >
    class basic_ofstream;
typedef basic_ofstream<char> ofstream;
typedef basic_ofstream<wchar_t> wofstream;
template<class E, class T = char_traits<E> >
    class basic_fstream;
typedef basic_fstream<char> fstream;
typedef basic_fstream<wchar_t> wfstream;
    }
```

Classes

basic_filebuf

Description: The template class describes a **stream buffer** that controls the transmission of elements of type E, whose character traits are determined by the class T, to and from a sequence of elements stored in an external file.

An object of class basic_filebuf<E, T> stores a **file pointer**, which designates the FILE object that controls the **stream** associated with an open file. It also stores pointers to two **file conversion facets** for use by the protected member functions overflow and underflow.

Synopsis:

```
template <class E, class T = char_traits<E> >
    class basic_filebuf : public basic_streambuf<E, T> {
public:
    typedef typename basic_streambuf<E, T>::char_type
        char_type;
    typedef typename basic_streambuf<E, T>::traits_type
        traits_type;
    typedef typename basic_streambuf<E, T>::int_type
        int_type;
    typedef typename basic_streambuf<E, T>::pos_type
        pos_type;
    typedef typename basic_streambuf<E, T>::off_type
        off_type;
    basic_filebuf();
    bool is_open() const;
    basic_filebuf *open(const char *s,
        ios_base::openmode mode);
    basic_filebuf *close();
protected:
    virtual pos_type seekoff(off_type off,
        ios_base::seekdir way,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type pos,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    virtual int_type underflow();
    virtual int_type pbackfail(int_type c =
        traits_type::eof());
```

```
    virtual int_type overflow(int_type c =
        traits_type::eof());
    virtual int sync();
    virtual basic_streambuf<E, T>
        *setbuf(E *s, streamsize n);
    };
```

Constructor:

basic_filebuf::basic_filebuf:

```
basic_filebuf();
```

The constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. It also stores a null pointer in the file pointer.

Types:

basic_filebuf::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_filebuf::int_type:

```
typedef typename traits_type::int_type int_type;
```

The type is a synonym for `traits_type::int_type`.

basic_filebuf::off_type:

```
typedef typename traits_type::off_type off_type;
```

The type is a synonym for `traits_type::off_type`.

basic_filebuf::pos_type:

```
typedef typename traits_type::pos_type pos_type;
```

The type is a synonym for `traits_type::pos_type`.

basic_filebuf::traits_type:

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

Member functions:

basic_filebuf::close:

```
basic_filebuf *close();
```

The member function returns a null pointer if the file pointer `fp` is a null pointer. Otherwise, it calls `fclose(fp)`. If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns `this` to indicate that the file was successfully closed.

For a wide stream, if any insertions have occured since the stream was opened, or since the last call to `streampos`, the function calls `overflow()`. It also inserts any sequence needed to restore the initial conversion state, by using the file conversion facet `fac` to call `fac.unshift` as needed. Each element `x` of type *char* thus produced is written to the associated stream designated by the file pointer `fp` as if by successive calls of the form `fputc(x, fp)`. If the call to `fac.unshift` or any write fails, the function does not succeed.

basic_filebuf::is_open:

```
bool is_open();
```

The member function returns true if the file pointer is not a null pointer.

basic_filebuf::open:

```
basic_filebuf *open(const char *s,
    ios_base::openmode mode);
```

The member function endeavors to open the file with filename `s`, by calling `fopen(s, strmode)`. Here `strmode` is determined from `mode & ~(ate & | binary)`:

- `ios_base::in` becomes `"r"` (open existing file for reading).
- `ios_base::out` or `ios_base::out | ios_base::trunc` becomes `"w"` (truncate existing file or create for writing).
- `ios_base::out | app` becomes `"a"` (open existing file for appending all writes).
- `ios_base::in | ios_base::out` becomes `"r+"` (open existing file for reading and writing).
- `ios_base::in | ios_base::out | ios_base::trunc` becomes `"w+"` (truncate existing file or create for reading and writing).
- `ios_base::in | ios_base::out | ios_base::app` becomes `"a+"` (open existing file for reading and for appending all writes).

If `mode & ios_base::binary` is nonzero, the function appends `b` to `strmode` to open a binary stream instead of a text stream. It then stores the value returned by `fopen` in the file pointer `fp`. If `mode & ios_base::ate` is nonzero and the file pointer is not a null pointer, the function calls `fseek(fp, 0, SEEK_END` to position the stream at end-of-file. If that positioning operation fails, the function calls `close(fp)` and stores a null pointer in the file pointer.

If the file pointer is not a null pointer, the function determines the **file conversion facet**: `use_facet< codecvt<E, char, traits_type:: state_type> >(getloc())`, for use by `underflow` and `overflow`.

If the file pointer is a null pointer, the function returns a null pointer. Otherwise, it returns `this`.

basic_filebuf::sync:

```
int sync();
```

The protected member function returns zero if the file pointer `fp` is a null pointer. Otherwise, it returns zero only if calls to both `overflow()` and `fflush(fp)` succeed in flushing any pending output to the stream.

Protected virtual member functions:

basic_filebuf::overflow:

```
virtual int_type overflow(int_type c =
    traits_type::eof());
```

If c != traits_type::eof(), the protected virtual member function endeavors to insert the element traits_type::to_char_type(c) into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer.
- It can convert any pending output in the output buffer, followed by c, by using the file conversion facet fac to call fac.out as needed. Each element x of type *char* thus produced is written to the associated stream designated by the file pointer fp as if by successive calls of the form fputc(x, fp). If any conversion or write fails, the function does not succeed.

If the function cannot succeed, it returns traits_type::eof(). Otherwise, it returns traits_type::not_eof(c).

basic_filebuf::pbackfail:

```
virtual int_type pbackfail(int_type c =
    traits_type::eof());
```

The protected virtual member function endeavors to put back an element into the input buffer, then make it the current element (pointed to by the next pointer). If c == traits_type::eof(), the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by x = traits_type::to_char_type(c). The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to x, it can simply decrement the next pointer for the input buffer.
- If the function can make a putback position available, it can do so, set the next pointer to point at that position, and store x in that position.
- If the function can push back an element onto the input stream, it can do so, such as by calling ungetc for an element of type *char.*

If the function cannot succeed, it returns traits_type::eof(). Otherwise, it returns traits_type::not_eof(c).

basic_filebuf::seekoff:

```
virtual pos_type seekoff(off_type off,
    ios_base::seekdir way,
    ios_base::openmode which =
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class basic_filebuf<E, T>, a stream position can be represented by an object of type fpos_t, which stores an offset and any state information needed to parse a wide stream. Offset zero designates the first element of the stream. (An object of type pos_type stores at least an fpos_t object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch between inserting and extracting, you must call either pubseekoff or pubseekpos. Calls to pubseekoff (and hence to seekoff) have various limitations for text streams, binary streams, and wide streams.

If the file pointer fp is a null pointer, the function fails. Otherwise, it endeavors to alter the stream position by calling fseek(fp, off, way). If that function succeeds and the resultant position fposn can be determined by calling fgetpos(fp, &fposn), the function succeeds. If the function succeeds, it returns a value of type pos_type containing fposn. Otherwise, it returns an invalid stream position.

basic_filebuf::seekpos:

```
virtual pos_type seekpos(pos_type pos,
    ios_base::openmode which =
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class basic_filebuf<E, T>, a stream position can be represented by an object of type fpos_t, which stores an offset and any state information needed to parse a wide stream. Offset zero designates the first element of the stream. (An object of type pos_type stores at least an fpos_t object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch between inserting and extracting, you must call either pubseekoff or pubseekpos. Calls to pubseekoff (and hence to seekoff) have various limitations for text streams, binary streams, and wide streams.

For a wide stream, if any insertions have occured since the stream was opened, or since the last call to streampos, the function calls overflow(). It also inserts any sequence needed to restore the initial conversion state, by using the file conversion facet fac to call fac.unshift as needed. Each element x of type *char* thus produced is written to the associated stream designated by the file pointer fp as if by successive calls of the form fputc(x, fp). If the call to fac.unshift or any write fails, the function does not succeed.

If the file pointer fp is a null pointer, the function fails. Otherwise, it endeavors to alter the stream position by calling fsetpos(fp, &fposn), where fposn is the fpos_t object stored in pos. If that function succeeds, the function returns pos. Otherwise, it returns an invalid stream position.

basic_filebuf::setbuf:

```
virtual basic_streambuf<E, T>
    *setbuf(E *s, streamsize n);
```

The protected member function returns zero if the file pointer fp is a null pointer. Otherwise, it calls setvbuf(fp, (char *)s, _IOFBF, n * sizeof (E)) to offer the array of n elements beginning at s as a buffer for the stream. If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns this to signal success.

basic_filebuf::underflow:

```
virtual int_type underflow();
```

The protected virtual member function endeavors to extract the current element c from the input stream, and return the element as traits_type::to_int_type(c). It can do so in various ways:

- If a read position is available, it takes c as the element stored in the read position and advances the next pointer for the input buffer.

- It can read one or more elements of type *char*, as if by successive calls of the form `fgetc(fp)`, and convert them to an element `c` of type `E` by using the file conversion facet `fac` to call `fac.in` as needed. If any read or conversion fails, the function does not succeed.

If the function cannot succeed, it returns `traits_type::eof()`. Otherwise, it returns `c`, converted as described above.

basic_fstream

Description: The template class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class `basic_filebuf<E, T>`, with elements of type `E`, whose character traits are determined by the class `T`. The object stores an object of class `basic_filebuf<E, T>`.

Synopsis:

```
template <class E, class T = char_traits<E> >
    class basic_fstream : public basic_iostream<E, T> {
public:
    basic_fstream();
    explicit basic_fstream(const char *s,
        ios_base::openmode mode =
            ios_base::in | ios_base::out);
    basic_filebuf<E, T> *rdbuf() const;
    bool is_open() const;
    void open(const char *s,
        ios_base::openmode mode =
            ios_base::in | ios_base::out);
    void close();
    };
```

Constructor:

basic_fstream::basic_fstream:

```
basic_fstream();
explicit basic_fstream(const char *s,
    ios_base::openmode mode =
        ios_base::in | ios_base::out);
```

The first constructor initializes the base class by calling `basic_iostream(sb)`, where `sb` is the stored object of class `basic_filebuf<E, T>`. It also initializes `sb` by calling `basic_filebuf<E, T>()`.

The second constructor initializes the base class by calling `basic_iostream(sb)`. It also initializes `sb` by calling `basic_filebuf<E, T>()`, then `sb.open(s, mode)`. If the latter function returns a null pointer, the constructor calls `setstate(failbit)`.

Member functions:

basic_fstream::close:

```
 voidclose();
```

The member function calls `rdbuf()-> close()`.

basic_fstream::is_open:

```
bool is_open();
```

The member function returns `rdbuf()-> is_open()`.

basic_fstream::open:

```
void open(const char *s,
    ios_base::openmode mode =
        ios_base::in | ios_base::out);
```

The member function calls `rdbuf()-> open(s, mode)`. If that function returns a null pointer, the function calls `setstate(failbit)`.

basic_fstream::rdbuf:

```
basic_filebuf<E, T> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `basic_filebuf<E, T>`.

basic_ifstream

Description: The template class describes an object that controls extraction of elements and encoded objects from a stream buffer of class `basic_filebuf<E, T>`, with elements of type `E`, whose character traits are determined by the class `T`. The object stores an object of class `basic_filebuf<E, T>`.

Synopsis:

```
template <class E, class T = char_traits<E> >
    class basic_ifstream : public basic_istream<E, T> {
public:
    basic_filebuf<E, T> *rdbuf() const;
    basic_ifstream();
    explicit basic_ifstream(const char *s,
        ios_base::openmode mode = ios_base::in);
    bool is_open() const;
    void open(const char *s,
        ios_base::openmode mode = ios_base::in);
    void close();
    };
```

Constructor:

basic_ifstream::basic_ifstream:

```
basic_ifstream();
explicit basic_ifstream(const char *s,
    ios_base::openmode mode = ios_base::in);
```

The first constructor initializes the base class by calling `basic_istream(sb)`, where `sb` is the stored object of class `basic_filebuf<E, T>`. It also initializes `sb` by calling `basic_filebuf<E, T>()`.

The second constructor initializes the base class by calling `basic_istream(sb)`. It also initializes `sb` by calling `basic_filebuf<E, T>()`, then `sb.open(s, mode | ios_base::in)`. If the latter function returns a null pointer, the constructor calls `setstate(failbit)`.

Member functions:

basic_ifstream::close:

```
void close();
```

The member function calls `rdbuf()-> close()`.

basic_ifstream::is_open:

```
bool is_open();
```

The member function returns `rdbuf()-> is_open()`.

basic_ifstream::open:

```
void open(const char *s,
    ios_base::openmode mode = ios_base::in);
```

The member function calls `rdbuf()-> open(s, mode | ios_base::in)`. If that function returns a null pointer, the function calls `setstate(failbit)`.

basic_ifstream::rdbuf:

```
basic_filebuf<E, T> *rdbuf() const
```

The member function returns the address of the stored stream buffer.

basic_ofstream

Description: The template class describes an object that controls insertion of elements and encoded objects into a stream buffer of class `basic_filebuf<E, T>`, with elements of type `E`, whose character traits are determined by the class `T`. The object stores an object of class `basic_filebuf<E, T>`.

Synopsis:

```
template <class E, class T = char_traits<E> >
    class basic_ofstream : public basic_ostream<E, T> {
public:
    basic_filebuf<E, T> *rdbuf() const;
    basic_ofstream();
    explicit basic_ofstream(const char *s,
        ios_base::openmode mode = ios_base::out);
    bool is_open() const;
    void open(const char *s,
        ios_base::openmode mode = ios_base::out);
    void close();
    };
```

Constructor:

basic_ofstream::basic_ofstream:

```
basic_ofstream();
explicit basic_ofstream(const char *s,
    ios_base::openmode which = ios_base::out);
```

The first constructor initializes the base class by calling `basic_ostream(sb)`, where `sb` is the stored object of class `basic_filebuf<E, T>`. It also initializes `sb` by calling `basic_filebuf<E, T>()`.

The second constructor initializes the base class by calling `basic_ostream(sb)`. It also initializes `sb` by calling `basic_filebuf<E, T>()`, then `sb.open(s, mode | ios_base::out)`. If the latter function returns a null pointer, the constructor calls `setstate(failbit)`.

Member functions:

basic_ofstream::close:

```
void close();
```

The member function calls `rdbuf()-> close()`.

basic_ofstream::is_open:

```
bool is_open();
```

The member function returns `rdbuf()-> is_open()`.

basic_ofstream::open:

```
void open(const char *s,
    ios_base::openmode mode = ios_base::out);
```

The member function calls `rdbuf()-> open(s, mode | ios_base::out)`. If that function returns a null pointer, the function calls `setstate(failbit)`.

basic_ofstream::rdbuf:

```
basic_filebuf<E, T> *rdbuf() const
```

The member function returns the address of the stored stream buffer.

Types

filebuf

```
typedef basic_filebuf<char, char_traits<char> > filebuf;
```

The type is a synonym for template class basic_filebuf, specialized for elements of type *char* with default character traits.

fstream

```
typedef basic_fstream<char, char_traits<char> > fstream;
```

The type is a synonym for template class basic_fstream, specialized for elements of type *char* with default character traits.

ifstream

```
typedef basic_ifstream<char, char_traits<char> > ifstream;
```

The type is a synonym for template class basic_ifstream, specialized for elements of type *char* with default character traits.

ofstream

```
typedef basic_ofstream<char, char_traits<char> >
    ofstream;
```

The type is a synonym for template class basic_ofstream, specialized for elements of type *char* with default character traits.

wfstream

```
typedef basic_fstream<wchar_t, char_traits<wchar_t> >
    wfstream;
```

The type is a synonym for template class basic_fstream, specialized for elements of type `wchar_t` with default character traits.

wifstream

```
typedef basic_ifstream<wchar_t, char_traits<wchar_t> >
    wifstream;
```

The type is a synonym for template class basic_ifstream, specialized for elements of type `wchar_t` with default character traits.

wofstream

```
typedef basic_ofstream<wchar_t, char_traits<wchar_t> >
    wofstream;
```

The type is a synonym for template class basic_ofstream, specialized for elements of type `wchar_t` with default character traits.

wfilebuf

```
typedef basic_filebuf<wchar_t, char_traits<wchar_t> >
    wfilebuf;
```

The type is a synonym for template class basic_filebuf, specialized for elements of type `wchar_t` with default character traits.

<functional>

Description

Include the STL standard header **<functional>** to define several templates that help construct **function objects**, objects of a type that defines `operator()`. A function object can thus be a function pointer, but in the more general case the object can store additional information that can be used during a function call.

Note: Additional functionality has been added to this header for TR1. To enable this functionality, you must define the macro __IBMCPP_TR1__ and use the **TARGET** compiler option to specify a valid release level. Valid release levels for TR1 support are zOSV1R12 or later. For example, you can specify the **TARGET** option as follows: `TARGET(zOSV1R12)`.

Any release prior to zOSV1R12 is invalid for the use of TR1. If TR1 code is used in an application to be compiled on an earlier platform, the compiler will issue the `#error` directive.

The following terminology applies to features added with TR1:

A **call signature** is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.

A **call wrapper** is an object of a call wrapper type.

A **call wrapper type** is a type that holds a callable object and supports a call operation that forwards to that object.

A **callable object** is an object of a callable type.

A **callable type** is a pointer to function, a pointer to member function, a pointer to member data, or a class type whose objects can appear immediately to the left of a function call operator.

A **target object** is the callable object held by a call wrapper object.

The pseudo-function `INVOKE(f, t1, t2, ..., tN)` means:

- `(t1.*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`;
- `((*t1).*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `t1` is not one of the types described in the previous item;
- `t1.*f` when `f` is a pointer to member data of class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`;
- `(*t1).*f` when `f` is a pointer to member data of class `T` and `t1` is not one of the types described in the previous item;
- `f(t1, t2, ..., tN)` in all other cases.

The pseudo-function `INVOKE(f, t1, t2, ..., tN, R)` means `INVOKE(f, t1, t2, ..., tN)` implicitly converted to `R`.

If a call wrapper has a **weak result type** the type of its member type `result_type` is based on the type `T` of the wrapper's target object:

- if `T` is a pointer to function, `result_type` is a synonym for the return type of `T`;
- if `T` is a pointer to member function, `result_type` is a synonym for the return type of `T`;
- if `T` is a pointer to data member, `result_type` is a synonym for the declared type of the data member;
- if `T` is a class type with a member type `result_type`, then `result_type` is a synonym for `T::result_type`;
- otherwise there is no member `result_type`.

Every call wrapper has a copy constructor. A **simple call wrapper** is a call wrapper that has an assignment operator and whose copy constructor and assignment operator do not throw exceptions. A **forwarding call wrapper** is a call wrapper that can be called with an argument list `t1, t2, ..., tN` where each `ti` is an lvalue.

The call wrappers defined in this header support function call operators with arguments of types `T1, T2, ..., TN`, where `0 <= N <= NMAX`. In this implementation the value of `NMAX` is 10. By default, the compiler generates code that enables function call operators with up to 3 arguments, thus, N = 3. When more arguments is required, _NARGS_CONST macro can be defined to value greater than 3. However, its value has to be within [0, 10]. This macro has to be used carefully because the compile time increases exponentially.

Synopsis

```
namespace std {
template<class Arg, class Result>
    struct unary_function;
template<class Arg1, class Arg2, class Result>
    struct binary_function;
template<class Ty>
    struct plus;
template<class Ty>
    struct minus;
template<class Ty>
    struct multiplies;
template<class Ty>
    struct divides;
template<class Ty>
    struct modulus;
```

```
template<class Ty>
    struct negate;
template<class Ty>
    struct equal_to;
template<class Ty>
    struct not_equal_to;
template<class Ty>
    struct greater;
template<class Ty>
    struct less;
template<class Ty>
    struct greater_equal;
template<class Ty>
    struct less_equal;
template<class Ty>
    struct logical_and;
template<class Ty>
    struct logical_or;
template<class Ty>
    struct logical_not;
template<class Fn1>
    struct unary_negate;
template<class Fn2>
    struct binary_negate;
template<class Fn2>
    class binder1st;
template<class Fn2>
    class binder2nd;
template<class Arg, class Result>
    class pointer_to_unary_function;
template<class Arg1, class Arg2, class Result>
    class pointer_to_binary_function;
template<class Result, class Ty>
    struct mem_fun_t;
template<class Result, class Ty, class Arg>
    struct mem_fun1_t;
template<class Result, class Ty>
    struct const_mem_fun_t;
template<class Result, class Ty, class Arg>
    struct const_mem_fun1_t;
template<class Result, class Ty>
    struct mem_fun_ref_t;
template<class Result, class Ty, class Arg>
    struct mem_fun1_ref_t;
template<class Result, class Ty>
    struct const_mem_fun_ref_t;
template<class Result, class Ty, class Arg>
    struct const_mem_fun1_ref_t;

// TEMPLATE FUNCTIONS
template<class Fn1>
    unary_negate<Fn1> not1(const Fn1& func);
template<class Fn2>
    binary_negate<Fn2> not2(const Fn2& func);
template<class Fn2, class Ty>
    binder1st<Fn2> bind1st(const Fn2& func, const Ty& left);
template<class Fn2, class Ty>
    binder2nd<Fn2> bind2nd(const Fn2& func, const Ty& left);
template<class Arg, class Result>
    pointer_to_unary_function<Arg, Result>
        ptr_fun(Result (*)(Arg));
template<class Arg1, class Arg2, class Result>
    pointer_to_binary_function<Arg1, Arg2, Result>
        ptr_fun(Result (*)(Arg1, Arg2));
template<class Result, class Ty>
    mem_fun_t<Result, Ty> mem_fun(Result (Ty::*pm)());
template<class Result, class Ty, class Arg>
```

```
    mem_fun1_t<Result, Ty, Arg> mem_fun(Result (Ty::*pm)(Arg left));
template<class Result, class Ty>
    const_mem_fun_t<Result, Ty> mem_fun(Result (Ty::*pm)() const);
template<class Result, class Ty, class Arg>
    const_mem_fun1_t<Result, Ty, Arg> mem_fun(Result (Ty::*pm)(Arg left) const);
template<class Result, class Ty>
    mem_fun_ref_t<Result, Ty> mem_fun_ref(Result (Ty::*pm)());
template<class Result, class Ty, class Arg>
    mem_fun1_ref_t<Result, Ty, Arg>
        mem_fun_ref(Result (Ty::*pm)(Arg left));
template<class Result, class Ty>
    const_mem_fun_ref_t<Result, Ty> mem_fun_ref(Result (Ty::*pm)() const);
template<class Result, class Ty, class Arg>
    const_mem_fun1_ref_t<Result, Ty, Arg>
        mem_fun_ref(Result (Ty::*pm)(Arg left) const);
#ifdef __IBMCPP_TR1__

namespace tr1 {
// TEMPLATE STRUCT hash
template <class Ty>
        struct hash;

// REFERENCE WRAPPERS
template <class Ty>
    reference_wrapper<Ty>
        ref(Ty&);
template <class Ty>
    reference_wrapper<Ty>
        ref(reference_wrapper<Ty>&);
template <class Ty>
    reference_wrapper<const Ty>
        cref(const Ty&);
template <class Ty>
    reference_wrapper<const Ty>
        cref(const reference_wrapper<Ty>&);
template <class Ty>
    struct reference_wrapper;

// FUNCTION OBJECT RETURN TYPES
template <class Ty>
    struct result_of;

// ENHANCED MEMBER POINTER ADAPTER
template <class Ret, class Ty>
    unspecified mem_fn(Ret Ty::*);

// FUNCTION OBJECT WRAPPERS
class bad_function_call;
template<class Fty>
    class function;
template<class Fty>
    void swap(function<Fty>& f1,
        function<Fty>& f2);
template<class Fty>
    bool operator!=(const function<Fty>&,
        null_ptr_type);
template<class Fty>
    bool operator!=(null_ptr_type,
        const function<Fty>&);
template<class Fty>
    bool operator==(const function<Fty>&,
        null_ptr_type);
template<class Fty>
    bool operator==(null_ptr_type,
        const function<Fty>&);

// ENHANCED BINDERS
```

```
template <class Fty, class T1, class T2, ..., class TN>
    unspecified bind(Fty, T1, T2, ..., TN);
template <class Ret, class Fty, class T1, class T2, ..., class TN>
    unspecified bind(Fty, T1, T2, ..., TN);
template <class Ret, class Ty, class T1, class T2, ..., class TN>
    unspecified bind(Ret Ty::*, T1, T2, ..., TN);

template <class Ty>
    struct is_bind_expression;
template <class Ty>
    struct is_placeholder;

namespace placeholders {
    extern unspecified _1;  // _2, _3, ... _M
        } // namespace placeholders
    } //namespace tr1
} //namespace std

#endif /* def __IBMCPP_TR1__ */
```

Classes

bad_function_call

```
class bad_function_call
    : public std::exception {
    };
```

[Added with TR1]

The class describes an exception thrown to indicate that a call to `operator()` on a "function" on page 92 object failed because the object was empty.

binary_function

```
template<class Arg1, class Arg2, class Result>
    struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
    };
```

The template class serves as a base for classes that define a member function of the form:

```
result_type operator()(const first_argument_type&,
    const second_argument_type&) const
```

or a similar form taking two arguments.

Hence, all such **binary functions** can refer to their first argument type as **first_argument_type**, their second argument type as **second_argument_type**, and their return type as **result_type**.

binary_negate

```
template<class Fn2>
    class binary_negate
        : public binary_function<
            typename Fn2::first_argument_type,
            typename Fn2::second_argument_type, bool> {
public:
    explicit binary_negate(const Fn2& func);
```

```
    bool operator()(
        const typename Fn2::first_argument_type& left,
        const typename Fn2::second_argument_type& right) const;
    };
```

The template class stores a copy of func, which must be a binary function object. It defines its member function **operator()** as returning !func(left, right).

binder1st

```
template<class Fn2>
    class binder1st
        : public unary_function<
            typename Fn2::second_argument_type,
            typename Fn2::result_type> {
public:
    typedef typename Fn2::second_argument_type argument_type;
    typedef typename Fn2::result_type result_type;
    binder1st(const Fn2& func,
        const typename Fn2::first_argument_type& left);
    result_type operator()(const argument_type& right) const;
protected:
    Fn2 op;
    typename Fn2::first_argument_type value;
    };
```

The template class stores a copy of func, which must be a binary function object, in **op**, and a copy of left in **value**. It defines its member function **operator()** as returning op(value, right).

binder2nd

```
template<class Fn2>
    class binder2nd
        : public unary_function<
            typename Fn2::first_argument_type,
            typename Fn2::result_type> {
public:
    typedef typename Fn2::first_argument_type argument_type;
    typedef typename Fn2::result_type result_type;
    binder2nd(const Fn2& func,
        const typename Fn2::second_argument_type& right);
    result_type operator()(const argument_type& left) const;
protected:
    Fn2 op;
    typename Fn2::second_argument_type value;
    };
```

The template class stores a copy of func, which must be a binary function object, in **op**, and a copy of right in **value**. It defines its member function **operator()** as returning op(left, value).

const_mem_fun_t

```
template<class Result, class Ty>
    struct const_mem_fun_t
        : public unary_function<const Ty *, Result> {
    explicit const_mem_fun_t(Result (Ty::*pm)() const);
    Result operator()(const Ty *pleft) const;
    };
```

The template class stores a copy of pm, which must be a pointer to a member function of class Ty, in a private member object. It defines its member function **operator()** as returning (pleft->*pm)() const.

const_mem_fun_ref_t

```
template<class Result, class Ty>
    struct const_mem_fun_ref_t
        : public unary_function<Ty, Result> {
    explicit const_mem_fun_t(Result (Ty::*pm)() const);
    Result operator()(const Ty& left) const;
    };
```

The template class stores a copy of pm, which must be a pointer to a member function of class Ty, in a private member object. It defines its member function **operator()** as returning (left.*Pm)() const.

const_mem_fun1_t

```
template<class Result, class Ty, class Arg>
    struct const_mem_fun1_t
        : public binary_function<Ty *, Arg, Result> {
    explicit const_mem_fun1_t(Result (Ty::*pm)(Arg) const);
    Result operator()(const Ty *pleft, Arg right) const;
    };
```

The template class stores a copy of pm, which must be a pointer to a member function of class Ty, in a private member object. It defines its member function **operator()** as returning (pleft->*pm)(right) const.

const_mem_fun1_ref_t

```
template<class Result, class Ty, class Arg>
    struct const_mem_fun1_ref_t
        : public binary_function<Ty, Arg, Result> {
    explicit const_mem_fun1_ref_t(Result (Ty::*pm)(Arg) const);
    Result operator()(const Ty& left, Arg right) const;
    };
```

The template class stores a copy of pm, which must be a pointer to a member function of class Ty, in a private member object. It defines its member function **operator()** as returning (left.*pm)(right) const.

divides

```
template<class Ty>
    struct divides : public binary_function<Ty, Ty, Ty> {
    Ty operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning left/ right.

equal_to

```
template<class Ty>
    struct equal_to
        : public binary_function<Ty, Ty, bool> {
    bool operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning left == right.

function

Description: [Added with TR1]

The template class is a call wrapper whose call signature is Ret(T1, T2, ..., TN).

Some member functions take an **operand** that names the desired target object. You can specify such an operand in several ways:

- `fn` — the callable object `fn`; after the call the `function` object holds a copy of `fn`
- `fnref` — the callable object named by `fnref.get()`; after the call the `function` object holds a reference to `fnref.get()`
- `right` — the callable object, if any, held by the `function` object `right`
- `npc` — a null pointer; after the call the `function` object is empty

In all cases, `INVOKE(f, t1, t2, ..., tN)`, where `f` is the callable object and `t1, t2, ..., tN` are lvalues of types `T1, T2, ..., TN` respectively, must be well-formed and, if `Ret` is not void, convertible to `Ret`.

An **empty** `function` object does not hold a callable object or a reference to a callable object.

Synopsis:

```
template <class Fty>
class function  // Fty of type Ret(T1, T2, ..., TN)
    : public unary_function<T1, Ret>       // when Fty is Ret(T1)
    : public binary_function<T1, T2, Ret>  // when Fty is Ret(T1, T2)
    {
public:
    typedef Ret result_type;

    function();
    function(null_ptr_type);
    function(const function&);
    template<class Fty2>
        function(Fty2);
    template<class Fty2>
        function(reference_wrapper<Fty2>);

    function& operator=(null_ptr_type);
    function& operator=(const function&);
    template<class Fty2>
        function& operator=(Fty2);
    template<class Fty2>
        function& operator=(reference_wrapper<Fty2>);
    void swap(function&);

    operator unspecified() const;
    result_type operator()(T1, T2, ....., TN) const;

    const std::type_info& target_type() const;
    template<class Fty2>
        Fty2 *target();
    template<class Fty2>
        const Fty2 *target() const;

private:
    template<class Fty2>
      bool operator==(const Fty2&) const;  // not defined
     template<class Fty2>
      bool operator!=(const Fty2&) const;  // not defined
      };
```

Constructor:

function::function:

```
function();
function(null_ptr_type npc);
function(const function& right);
```

```
template<class F>
    function(Fty fn);
template<class F>
    function(reference_wrapper<Fty> fnref);
```

The first two constructors construct an empty `function` object. The other constructors construct a `function` object that holds the callable object passed as the operand.

Types:

function::result_type:

```
typedef Ret result_type;
```

The typedef is a synonym for the type `Ret` in the template's call signature.

function::target:

```
template<class Fty2> Fty2 *target();
template<class Fty2> const Fty2 *target() const;
```

The type `Fty2` must be callable for the argument types `T1, T2, ..., TN` and the return type `Ret`. If `target_type() == typeid(Fty2)`, the member template function returns the address of the target object; otherwise, it returns 0.

A type `Fty2` is **callable** for the argument types `T1, T2, ..., TN` and the return type `Ret` if, for lvalues `fn, t1, t2, ..., tN` of types `Fty2, T1, T2, ..., TN`, respectively, `INVOKE (fn, t1, t2, ..., tN)` is well-formed and, if `Ret` is not `void`, convertible to `Ret`.

Member functions:

function::operator():

```
result_type operator()(T1 t1, T2 t2, ..., TN tN);
```

The member function returns `INVOKE (fn, t1, t2, ..., tN, Ret)`, where `fn` is the target object stored in `*this`.

function::swap:

```
void swap(function& right);
```

The member function swaps the target objects between `*this` and `right`. It does so in constant time and throws no exceptions.

function::target_type:

```
const std::type_info& target_type() const;
```

The member function returns `typeid(void)` if `*this` is empty, otherwise it returns `typeid(T)`, where `T` is the type of the target object.

Operators:

function::operator=:

```
function& operator=(null_ptr_type npc);
function& operator=(const function& right);
template<class Fty>
```

```
    function& operator=(Fty fn);
template<class Fty>
    function& operator=(reference_wrapper<Fty> fnref);
```

The operators each replace the callable object held by `*this` with the callable object passed as the operand.

function::operator==:

```
template<class Fty2>
    bool operator==(const function<Fty2>&);
```

The operator is private so that it cannot be called.

function::operator!=:

```
template<class Fty2>
    bool operator!=(const function<Fty2>&);
```

The operator is private so that it cannot be called.

function::operator unspecified:

```
operator unspecified();
```

The operator returns a value that is convertible to `bool` with a true value only if the object is not empty.

greater

```
template<class Ty>
    struct greater : public binary_function<Ty, Ty, bool> {
    bool operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning `left > right`. The member function defines a total ordering, even if `Ty` is an object pointer type.

greater_equal

```
template<class Ty>
    struct greater_equal
        : public binary_function<Ty, Ty, bool> {
    bool operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning `left >= right`. The member function defines a total ordering if `Ty` is an object pointer type.

hash

```
template <class Ty>
    struct hash
        : public unary_function<Ty, size_t> {
    size_t operator()(Ty val) const;
    };
```

The template class defines its member function as returning a value uniquely determined by val. The member function defines a **hash function**, suitable for mapping values of type `Ty` to a distribution of index values. `Ty` may be any scalar type, string, wstring, or, beginning with C++0x, u16string, or u32string.

less

```
template<class Ty>
    struct less : public binary_function<Ty, Ty, bool> {
    bool operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning `left < right`. The member function defines a total ordering if `Ty` is an object pointer type.

less_equal

```
template<class Ty>
    struct less_equal
        : public binary_function<Ty, Ty, bool> {
    bool operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning `left <= right`. The member function defines a total ordering, even if `Ty` is an object pointer type.

logical_and

```
template<class Ty>
    struct logical_and
        : public binary_function<Ty, Ty, bool> {
    bool operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning `left && right`.

logical_not

```
template<class Ty>
    struct logical_not : public unary_function<Ty, bool> {
    bool operator()(const Ty& left) const;
    };
```

The template class defines its member function as returning `!left`.

logical_or

```
template<class Ty>
    struct logical_or
        : public binary_function<Ty, Ty, bool> {
    bool operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning `left || right`.

mem_fun_t

```
template<class Result, class Ty>
    struct mem_fun_t : public unary_function<Ty *, Result> {
    explicit mem_fun_t(Result (Ty::*pm)());
    Result operator()(Ty *pleft) const;
    };
```

The template class stores a copy of `pm`, which must be a pointer to a member function of class `Ty`, in a private member object. It defines its member function **operator()** as returning `(pleft->*pm)()`.

mem_fun_ref_t

```
template<class Result, class Ty>
    struct mem_fun_ref_t
        : public unary_function<Ty, Result> {
    explicit mem_fun_t(Result (Ty::*pm)());
    Result operator()(Ty& left) const;
    };
```

The template class stores a copy of pm, which must be a pointer to a member function of class Ty, in a private member object. It defines its member function **operator()** as returning (left.*Pm)().

mem_fun1_t

```
template<class Result, class Ty, class Arg>
    struct mem_fun1_t
        : public binary_function<Ty *, Arg, Result> {
    explicit mem_fun1_t(Result (Ty::*pm)(Arg));
    Result operator()(Ty *pleft, Arg right) const;
    };
```

The template class stores a copy of pm, which must be a pointer to a member function of class Ty, in a private member object. It defines its member function **operator()** as returning (pleft->*pm)(right).

mem_fun1_ref_t

```
template<class Result, class Ty, class Arg>
    struct mem_fun1_ref_t
        : public binary_function<Ty, Arg, Result> {
    explicit mem_fun1_ref_t(Result (Ty::*pm)(Arg));
    Result operator()(Ty& left, Arg right) const;
    };
```

The template class stores a copy of pm, which must be a pointer to a member function of class Ty, in a private member object. It defines its member function **operator()** as returning (left.*pm)(right).

minus

```
template<class Ty>
    struct minus : public binary_function<Ty, Ty, Ty> {
    Ty operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning left - right.

modulus

```
template<class Ty>
    struct modulus : public binary_function<Ty, Ty, Ty> {
    Ty operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning left % right.

multiplies

```
template<class Ty>
    struct multiplies : public binary_function<Ty, Ty, Ty> {
    Ty operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning left * right.

negate

```
template<class Ty>
    struct negate : public unary_function<Ty, Ty> {
    Ty operator()(const Ty& left) const;
    };
```

The template class defines its member function as returning `-left`.

not_equal_to

```
template<class Ty>
    struct not_equal_to
        : public binary_function<Ty, Ty, bool> {
    bool operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning `left != right`.

plus

```
template<class Ty>
    struct plus : public binary_function<Ty, Ty, Ty> {
    Ty operator()(const Ty& left, const Ty& right) const;
    };
```

The template class defines its member function as returning `left + right`.

pointer_to_binary_function

```
template<class Arg1, class Arg2, class Result>
    class pointer_to_binary_function
        : public binary_function<Arg1, Arg2, Result> {
public:
    explicit pointer_to_binary_function(
        Result (*pfunc)(Arg1, Arg2));
    Result operator()(const Arg1 left, const Arg2 right) const;
    };
```

The template class stores a copy of `pfunc`. It defines its member function **operator()** as returning `(*pfunc)(left, right)`.

pointer_to_unary_function

```
template<class Arg, class Result>
    class pointer_to_unary_function
        : public unary_function<Arg, Result> {
public:
    explicit pointer_to_unary_function(
        Result (*pfunc)(Arg));
    Result operator()(const Arg left) const;
    };
```

The template class stores a copy of `pfunc`. It defines its member function **operator()** as returning `(*pfunc)(left)`.

reference_wrapper

Description: [Added with TR1]

A `reference_wrapper<Ty>` is copy constructible and assignable, and holds a pointer that points to an object of type `Ty`.

A specialization `reference_wrapper<Ty>` is derived from `std::unary_function<T1, Ret>` (hence defining the nested type `result_type` as a synonym for `Ret` and the nested type `argument_type` as a synonym for `T1`) only if the type `Ty` is:

- a function type or pointer to function type taking one argument of type `T1` and returning `Ret`; or
- a pointer to a member function `Ret T::f() cv`, where *cv* represents the member function's cv-qualifiers; the type `T1` is *cv* `T*`; or
- a class type that is derived from `unary_function<T1, Ret>`.

A specialization `reference_wrapper<Ty>` is derived from `std::binary_function<T1, T2, Ret>` (hence defining the nested type `result_type` as a synonym for `Ret`, the nested type `first_argument_type` as a synonym for `T1`, and the nested type `second_argument_type` as a synonym for `T2`) only if the type `Ty` is:

- a function type or pointer to function type taking two arguments of types `T1` and `T2` and returning `Ret`; or
- a pointer to a member function `Ret T::f(T2) cv`, where *cv* represents the member function's cv-qualifiers; the type `T1` is *cv* `T*`; or
- a class type that is derived from `binary_function<T1, T2, Ret>`.

Synopsis:

```
template<class Ty>
    class reference_wrapper
    : public unary_function<T1, Ret>        // see below
    : public binary_function<T1, T2, Ret>   // see below
    {
public:
    typedef Ty type;
    typedef T0 result_type;                 // see below

    explicit reference_wrapper(Ty&);

    Ty& get() const;
    operator Ty&() const;
    template<class T1, class T2, ..., class TN>
        typename result_of<T(T1, T2, ..., TN)>::type
        operator()(T1&, T2&, ..., TN&);

private:
    Ty *ptr; // exposition only
    };
```

Constructor:

reference_wrapper::reference_wrapper:

```
explicit reference_wrapper(Ty& val);
```

The constructor sets the stored value `ptr` to `&val`.

Types:

reference_wrapper::result_type:

```
typedef T0 result_type;
```

The typedef is a synonym for the (weak result type) of a wrapped callable object.

reference_wrapper::type:

```
typedef Ty type;
```

The typedef is a synonym for the template argument Ty.

Member functions:

reference_wrapper::get:

```
Ty& get() const;
```

The member function returns INVOKE (get(), t1, t2, ..., tN).

Operators:

reference_wrapper::operator ():

```
template<class T1, class T2, ..., class TN>
    typename result_of<T(T1, T2, ..., TN)>::type
    operator()(T1& t1, T2& t2, ..., TN& tN);
```

The template member operator returns INVOKE (get(), t1, t2, ..., tN).

reference_wrapper::operator Ty&:

```
operator Ty&() const;
```

The member operator returns *ptr.

result_of

```
template<class Ty>
    struct result_of {
    typedef T0 type;
    };
```

[Added with TR1]

The template class defines its member type as a synonym for the return type of a function call described by its template argument Ty. The template argument must be of the form Fty(T1, T2, ..., TN), where Fty is a callable type. The template determines the return type according to the first of the following rules that applies:

- if Fty is a pointer to function type *R(*)(U1, U2, ..., UN)* the return type is *R*;
- if Fty is a pointer to member function type *R(U1::*)(U2, ..., UN)* the return type is *R*;
- if Fty is a pointer to data member type *R U1::** the return type is *R*;
- if Fty is a class with a member typedef *result_type* the return type is *Fty::result_type*;
- if *N* is 0 (that is, Ty is of the form Fty()) the return type is *void*;
- If Fty is a class with a member template named *result* the return type is *Fty::result<T1, T2, ..., TN>::type*;
- in all other cases it is an error.

unary_function

```
template<class Arg, class Result>
    struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
    };
```

The template class serves as a base for classes that define a member function of the form:

```
result_type operator()(const argument_type&) const
```

or a similar form taking one argument.

Hence, all such **unary functions** can refer to their sole argument type as **argument_type** and their return type as **result_type**.

unary_negate

```
template<class Fn1>
    class unary_negate
        : public unary_function<
            typename Fn1::argument_type,
            bool> {
public:
    explicit unary_negate(const Fn1& Func);
    bool operator()(
        const typename Fn1::argument_type& left) const;
    };
```

The template class stores a copy of func, which must be a unary function object. It defines its member function **operator()** as returning !func(left).

Functions

bind

```
template <class Fty, class T1, class T2, ..., class TN>
   unspecified bind(Fty fn, T1 t1, T2 t2, ..., TN tN);
template <class Ret, class Fty, class T1, class T2, ..., class TN>
   unspecified bind(Fty fn, T1 t1, T2 t2, ..., TN tN);
```

[Added with TR1]

The types Fty, T1, T2, ..., TN must be copy constructible, and INVOKE(fn, t1, ..., tN) must be a valid expression for some values w1, w2, ..., wN.

The first template function returns a forwarding call wrapper g with a weak result type. The effect of g(u1, u2, ..., uM) is INVOKE(f, v1, v2, ..., vN, "result_of" on page 100<Fty *cv* (V1, V2, ..., VN)>::type), where *cv* is the cv-qualifiers of g and the values and types of the bound arguments v1, v2, ..., vN are determined as specified below.

The second template function returns a forwarding call wrapper g with a nested type result_type that is a synonym for Ret. The effect of g(u1, u2, ..., uM) is INVOKE(f, v1, v2, ..., vN, Ret), where *cv* is the cv-qualifiers of g and the values and types of the bound arguments v1, v2, ..., vN are determined as specified below.

The values of the **bound arguments** v1, v2, ..., vN and their corresponding types V1, V2, ..., VN depend on the type of the corresponding argument ti of type Ti in the call to bind and the cv-qualifiers *cv* of the call wrapper g as follows:

- if ti is of type reference_wrapper<T> the argument vi is ti.get() and its type Vi is T&;
- if the value of std::tr1::is_bind_expression<Ti>::value is true the argument vi is ti(u1, u2, ..., uM) and its type Vi is result_of<Ti cv (U1&, U2&, ..., UN&>::type;
- if the value j of std::tr1::is_placeholder<Ti>::value is not zero the argument vi is uj and its type Vi is Uj&;
- otherwise the argument vi is ti and its type Vi is Ti cv &.

For example, given a function `f(int, int)` the expression `bind(f, _1, 0)` returns a forwarding call wrapper `cw` such that `cw(x)` calls `f(x, 0)`. The expression `bind(f, 0, _1)` returns a forwarding call wrapper `cw` such that `cw(x)` calls `f(0, x)`.

The number of arguments in a call to `bind` in addition to the argument `fn` must be equal to the number of arguments that can be passed to the callable object `fn`. Thus, `bind(cos, 1.0)` is correct, and both `bind(cos)` and `bind(cos, _1, 0.0)` are incorrect.

The number of arguments in the function call to the call wrapper returned by `bind` must be at least as large as the highest numbered value of `is_placeholder<PH>::value` for all of the placeholder arguments in the call to `bind`. Thus, `bind(cos, _2)(0.0, 1.0)` is correct (and returns `cos(1.0)`), and `bind(cos, _2)(0.0)` is incorrect.

bind1st

```
template<class Fn2, class Ty>
    binder1st<Fn2> bind1st(const Fn2& func, const Ty& left);
```

The function returns `binder1st<Fn2>(func, typename Fn2::first_argument_type(left))`.

bind2nd

```
template<class Fn2, class Ty>
    binder2nd<Fn2> bind2nd(const Fn2& func, const Ty& right);
```

The function returns `binder2nd<Fn2>(func, typename Fn2::second_argument_type(right))`.

cref

```
template <class Ty>
    reference_wrapper<const Ty> cref(const Ty& arg);
template <class Ty>
    reference_wrapper<const Ty> cref(const reference_wrapper<Ty>& arg);
```

[Added with TR1]

The first function returns `reference_wrapper<const Ty>(arg.get())`. The second function returns `reference_wrapper<const Ty>(arg)`.

mem_fn

```
template <class Ret, class Ty>
    unspecified mem_fn(Ret Ty::*pm);
```

[Added with TR1]

The template function returns a simple call wrapper `cw`, with a weak result type, such that the expression `cw(t, a2, ..., aN)` is equivalent to `INVOKE (pm, t, a2, ..., aN)`. It does not throw any exceptions.

The returned call wrapper is derived from `std::unary_function<cv Ty*, Ret>` (hence defining the nested type `result_type` as a synonym for `Ret` and the nested type `argument_type` as a synonym for `cv Ty*`) only if the type `Ty` is a pointer to member function with cv-qualifier *cv* that takes no arguments.

The returned call wrapper is derived from `std::binary_function<cv Ty*, T2, Ret>` (hence defining the nested type `result_type` as a synonym for `Ret`, the nested type first argument_type as a synonym for cv Ty*, and the nested type second argument_type as a synonym for T2) only if the type Ty is a pointer to member function with cv-qualifier *cv* that takes one argument, of type T2.

mem_fun

```
template<class Result, class Ty>
    mem_fun_t<Result, Ty> mem_fun(Result (Ty::*pm)());
template<class Result, class Ty, class Arg>
    mem_fun1_t<Result, Ty, Arg> mem_fun(Result (Ty::*pm)(Arg));
template<class Result, class Ty>
    const_mem_fun_t<Result, Ty> mem_fun(Result (Ty::*pm)() const);
template<class Result, class Ty, class Arg>
    const_mem_fun1_t<Result, Ty, Arg> mem_fun(Result (Ty::*pm)(Arg) const);
```

The template function returns pm cast to the return type.

mem_fun_ref

```
template<class Result, class Ty>
    mem_fun_ref_t<Result, Ty> mem_fun_ref(Result (Ty::*pm)());
template<class Result, class Ty, class Arg>
    mem_fun1_ref_t<Result, Ty, Arg> mem_fun_ref(Result (Ty::*pm)(Arg));
template<class Result, class Ty>
    const_mem_fun_ref_t<Result, Ty> mem_fun_ref(Result (Ty::*pm)() const);
template<class Result, class Ty, class Arg>
    const_mem_fun1_ref_t<Result, Ty, Arg> mem_fun_ref(Result (Ty::*pm)(Arg) const);
```

The template function returns pm cast to the return type.

not1

```
template<class Fn1>
    unary_negate<Fn1> not1(const Fn1& func);
```

The template function returns unary_negate<Fn1>(func).

not2

```
template<class Fn2>
    binary_negate<Fn2> not2(const Fn2& func);
```

The template function returns binary_negate<Fn2>(func).

ptr_fun

```
template<class Arg, class Result>
    pointer_to_unary_function<Arg, Result>
        ptr_fun(Result (*pfunc)(Arg));
template<class Arg1, class Arg2, class Result>
    pointer_to_binary_function<Arg1, Arg2, Result>
        ptr_fun(Result (*pfunc)(Arg1, Arg2));
```

The first template function returns pointer_to_unary_function<Arg, Result>(pfunc).

The second template function returns pointer_to_binary_function<Arg1, Arg2, Result>(pfunc).

ref

```
template<class Ty>
    reference_wrapper<Ty> ref(Ty& arg);
template<class Ty>
    reference_wrapper<Ty> ref(reference_wrapper<Ty>& arg);
```

[Added with TR1]

The first function returns `reference_wrapper<Ty>(arg.get())`. The second function returns `reference_wrapper<Ty>(arg)`.

swap

```
template <class Fty>
void swap(function<Fty>& f1,
    function<Fty>& f2);
```

[Added with TR1]

The function returns `f1.swap(f2)`.

Operators

operator!=

```
template<class Fty>
    bool operator!=(const function<Fty>& f, null_ptr_type npc);
template<class Fty>
    bool operator!=(null_ptr_type npc, const function<Fty>& f);
```

[Added with TR1]

The operators both take an argument that is a reference to a `function` object and an argument that is a null pointer. Both return true only if the `function` object is not empty.

operator==

```
template<class Fty>
    bool operator==(const function<Fty>& f, null_ptr_type npc);
template<class Fty>
    bool operator==(null_ptr_type npc, const function<Fty>& f);
```

[Added with TR1]

The operators both take an argument that is a reference to a `function` object and an argument that is a null pointer. Both return true only if the `function` object is empty.

Structures

is_bind_expression

```
template <class Ty>
    struct is_bind_expression {
    static const bool value;
    };
```

[Added with TR1]

The constant value `value` is true if the type `Ty` is a type returned by a call to `bind`, otherwise false.

is_placeholder

```
template <class Ty>
    struct is_placeholder {
    static const int value;
    };
```

[Added with TR1]

The constant value `value` is 0 if the type `Ty` is not a placeholder; otherwise, its value is the position of the function call argument that it binds to.

Objects

_1

```
namespace placeholders {
  extern unspecified _1;  // _2, _3, ... _M
  } // namespace placeholders (within std::tr1)
```

[Added with TR1]

The objects `_1, _2, ... _M` are placeholders designating the first, second, ..., Mth argument, respectively in a function call to an object returned by "bind" on page 101. In this implementation the value of `M` is 10.

<iomanip>

Description

Include the iostreams standard header **<iomanip>** to define several manipulators that each take a single argument. Each of these manipulators returns an unspecified type, called `T1` through `T6` here, that overloads both `basic_istream<E, T>::operator>>` and `basic_ostream<E, T>::operator<<`. Thus, you can write extractors and inserters such as:

```
cin >> setbase(8);
cout << setbase(8);
```

Synopsis

```
namespace std {
T1 resetiosflags(ios_base::fmtflags mask);
T2 setiosflags(ios_base::fmtflags mask);
T3 setbase(int base);
template<class E>
    T4 setfill(E c);
T5 setprecision(streamsize n);
T6 setw(streamsize n);
    }
```

Manipulators

resetiosflags

```
T1 resetiosflags(ios_base::fmtflags mask);
```

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.setf(ios_base:: fmtflags(), mask)`, then returns `str`.

setbase

```
T3 setbase(int base);
```

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.setf(mask, ios_base::basefield)`, then returns `str`. Here, `mask` is determined as follows:

- If `base` is 8, then `mask` is `ios_base::oct`

- If base is 10, then mask is ios_base::dec
- If base is 16, then mask is ios_base::hex
- If base is any other value, then mask is ios_base::fmtflags(0)

setfill

```
template<class E>
    T4 setfill(E fillch);
```

The template manipulator returns an object that, when extracted from or inserted into the stream str, calls str.fill(fillch), then returns str. The type E must be the same as the element type for the stream str.

setiosflags

```
T2 setiosflags(ios_base::fmtflags mask);
```

The manipulator returns an object that, when extracted from or inserted into the stream str, calls str.setf(mask), then returns str.

setprecision

```
T5 setprecision(streamsize prec);
```

The manipulator returns an object that, when extracted from or inserted into the stream str, calls str.precision(prec), then returns str.

setw

```
T6 setw(streamsize wide);
```

The manipulator returns an object that, when extracted from or inserted into the stream str, calls str.width(wide), then returns str.

<ios>

Description

Include the iostreams standard header **<ios>** to define several types and functions basic to the operation of iostreams. (This header is typically included for you by another of the iostreams headers. You seldom have occasion to include it directly.)

A large group of functions are **manipulators**. A manipulator declared in <ios> alters the values stored in its argument object of class ios_base. Other manipulators perform actions on streams controlled by objects of a type derived from this class, such as a specialization of one of the template classes basic_istream or basic_ostream. For example, noskipws(str) clears the format flag ios_base::skipws in the object str, which might be of one of these types.

You can also call a manipulator by inserting it into an output stream or extracting it from an input stream, thanks to some special machinery supplied in the classes derived from ios_base. For example:

```
istr >> noskipws;
```

calls noskipws(istr).

Synopsis

```
namespace std {
typedef T1 streamoff;
typedef T2 streamsize;
class ios_base;

        // TEMPLATE CLASSES
template <class E, class T = char_traits<E> >
    class basic_ios;
typedef basic_ios<char, char_traits<char> > ios;
typedef basic_ios<wchar_t, char_traits<wchar_t> >
    wios;
template <class St>
    class fpos;
typedef fpos<mbstate_t> streampos;
typedef fpos<mbstate_t> wstreampos;

        // MANIPULATORS
ios_base& boolalpha(ios_base& str);
ios_base& noboolalpha(ios_base& str);
ios_base& showbase(ios_base& str);
ios_base& noshowbase(ios_base& str);
ios_base& showpoint(ios_base& str);
ios_base& noshowpoint(ios_base& str);
ios_base& showpos(ios_base& str);
ios_base& noshowpos(ios_base& str);
ios_base& skipws(ios_base& str);
ios_base& noskipws(ios_base& str);
ios_base& unitbuf(ios_base& str);
ios_base& nounitbuf(ios_base& str);
ios_base& uppercase(ios_base& str);
ios_base& nouppercase(ios_base& str);
ios_base& internal(ios_base& str);
ios_base& left(ios_base& str);
ios_base& right(ios_base& str);
ios_base& dec(ios_base& str);
ios_base& hex(ios_base& str);
ios_base& oct(ios_base& str);
ios_base& fixed(ios_base& str);
ios_base& scientific(ios_base& str);

namespace tr1 {
    ios_base& hexfloat(ios_base& str);
} // namespace tr1
} // namespace std
```

Classes

basic_ios

Description: The template class describes the storage and member functions common to both input streams (of template class basic_istream) and output streams (of template class basic_ostream) that depend on the template parameters. (The class ios_base describes what is common and *not* dependent on template parameters.) An object of class `basic_ios<E, T>` helps control a stream with elements of type `E`, whose character traits are determined by the class `T`.

An object of class `basic_ios<E, T>` stores:

- a **tie pointer** to an object of type `basic_ostream<E, T>`
- a **stream buffer pointer** to an object of type `basic_streambuf<E, T>`
- formatting information
- stream state information in a base object of type ios_base

- a **fill character** in an object of type *char_type*

Synopsis:

```
template <class E, class T = char_traits<E> >
    class basic_ios : public ios_base {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef typename T::int_type int_type;
    typedef typename T::pos_type pos_type;
    typedef typename T::off_type off_type;
    explicit basic_ios(basic_streambuf<E, T> *sb);
    virtual ~basic_ios();
    operator void *() const;
    bool operator!() const;
    iostate rdstate() const;
    void clear(iostate state = goodbit);
    void setstate(iostate state);
    bool good() const;
    bool eof() const;
    bool fail() const;
    bool bad() const;
    iostate exceptions() const;
    iostate exceptions(iostate except);
    basic_ios& copyfmt(const basic_ios& rhs);
    locale imbue(const locale& loc);
    char_type widen(char ch);
    char narrow(char_type ch, char dflt);
    char_type fill() const;
    char_type fill(char_type ch);
    basic_ostream<E, T> *tie() const;
    basic_ostream<E, T> *tie(basic_ostream<E, T> *str);
    basic_streambuf<E, T> *rdbuf() const;
    basic_streambuf<E, T>
        *rdbuf(basic_streambuf<E, T> *sb);
    E widen(char ch);
    char narrow(E ch, char dflt);
protected:
    void init(basic_streambuf<E, T> *sb);
    basic_ios();
    basic_ios(const facet&);     // not defined
    void operator=(const facet&) // not defined
        };
```

Constructor:

basic_ios::basic_ios:

```
explicit basic_ios(basic_streambuf<E, T> *sb);
basic_ios();
```

The first constructor initializes its member objects by calling `init(sb)`. The second (protected) constructor leaves its member objects uninitialized. A later call to `init` *must* initialize the object before it can be safely destroyed.

Types:

basic_ios::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_ios::int_type:

```
typedef typename T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_ios::off_type:

```
typedef typename T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_ios::pos_type:

```
typedef typename T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

basic_ios::traits_type:

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

Member functions:

basic_ios::bad:

```
bool bad() const;
```

The member function returns true if `rdstate() & badbit` is nonzero.

basic_ios::clear:

```
void clear(iostate state = goodbit);
```

The member function replaces the stored stream state information with `state | (rdbuf() != 0 ? goodbit : badbit)`. If `state & exceptions()` is nonzero, it then throws an object of class failure.

basic_ios::copyfmt:

```
basic_ios& copyfmt(const basic_ios& rhs);
```

The member function reports the callback event erase_event. It then copies from `rhs` into `*this` the fill character, the tie pointer, and the formatting information. Before altering the exception mask, it reports the callback event copyfmt_event. If, after the copy is complete, `state & exceptions()` is nonzero, the function effectively calls clear with the argument `rdstate()`. It returns `*this`.

basic_ios::eof:

```
bool eof() const;
```

The member function returns true if `rdstate() & eofbit` is nonzero.

basic_ios::exceptions:

```
iostate exceptions() const;
iostate exceptions(iostate except);
```

The first member function returns the stored exception mask. The second member function stores `except` in the exception mask and returns its previous stored value. Note that storing a new exception mask can throw an exception just like the call `clear( rdstate())`.

basic_ios::fail:

```
bool fail() const;
```

The member function returns true if `rdstate() & failbit` is nonzero.

basic_ios::fill:

```
char_type fill() const;
char_type fill(char_type ch);
```

The first member function returns the stored fill character. The second member function stores `ch` in the fill character and returns its previous stored value.

basic_ios::good:

```
bool good() const;
```

The member function returns true if `rdstate() == goodbit` (no state flags are set).

basic_ios::imbue:

```
locale imbue(const locale& loc);
```

If rdbuf is not a null pointer, the member function calls `rdbuf()->pubimbue(loc)`. In any case, it returns `ios_base::imbue(loc)`.

basic_ios::init:

```
void init(basic_streambuf<E, T> *sb);
```

The member function stores values in all member objects, so that:
- `rdbuf()` returns `sb`
- `tie()` returns a null pointer
- `rdstate()` returns `goodbit` if `sb` is nonzero; otherwise, it returns `badbit`
- `exceptions()` returns `goodbit`
- `flags()` returns `skipws | dec`
- `width()` returns zero
- `precision()` returns 6
- `fill()` returns the space character
- `getloc()` returns `locale::classic()`
- `iword` returns zero and `pword` returns a null pointer for all argument value

basic_ios::narrow:

```
char narrow(char_type ch, char dflt);
```

The member function returns `use_facet< ctype<E> >( getloc()). narrow(ch, dflt)`.

basic_ios::rdbuf:

```
basic_streambuf<E, T> *rdbuf() const;
basic_streambuf<E, T> *rdbuf(basic_streambuf<E, T> *sb);
```

The first member function returns the stored `stream buffer pointer`.

The second member function stores `sb` in the stored `stream buffer pointer` and returns the previously stored value.

basic_ios::rdstate:

```
iostate rdstate() const;
```

The member function returns the stored `stream state information`.

basic_ios::setstate:

```
void setstate(iostate state);
```

The member function effectively calls `clear(state | rdstate())`.

basic_ios::tie:

```
basic_ostream<E, T> *tie() const;
basic_ostream<E, T> *tie(basic_ostream<E, T> *str);
```

The first member function returns the stored tie pointer. The second member function stores `str` in the tie pointer and returns its previous stored value.

basic_ios::widen:

```
char_type widen(char ch);
```

The member function returns `use_facet< ctype<E> >( getloc()). widen(ch)`.

Operators:

*basic_ios::operator void *:*

```
operator void *() const;
```

The operator returns a null pointer only if `fail()`.

basic_ios::operator!:

```
bool operator!() const;
```

The operator returns `fail()`.

fpos

Description: The template class describes an object that can store all the information needed to restore an arbitrary file-position indicator within any stream. An object of class `fpos<St>` effectively stores at least two member objects:

- a byte offset, of type streamoff
- a conversion state, for use by an object of class `basic_filebuf`, of type **St**, typically mbstate_t

It can also store an arbitrary file position, for use by an object of class basic_filebuf, of type fpos_t. For an environment with limited file size, however, `streamoff` and `fpos_t` may sometimes be used interchangeably. And for an environment with no streams that have a state-dependent encoding, `mbstate_t` may actually be unused. So the number of member objects stored may vary.

Synopsis:

```
template <class St>
    class fpos {
public:
    fpos(streamoff off);
    explicit fpos(St state);
    St state() const;
```

```
    void state(St state);
    operator streamoff() const;
    streamoff operator-(const fpos& rhs) const;
    fpos& operator+=(streamoff off);
    fpos& operator-=(streamoff off);
    fpos operator+(streamoff off) const;
    fpos operator-(streamoff off) const;
    bool operator==(const fpos& rhs) const;
    bool operator!=(const fpos& rhs) const;
    };
```

Constructor:

fpos::fpos:

```
fpos(streamoff off);
explicit fpos(St state);
```

The first constructor stores the offset `off`, relative to the beginning of file and in the initial conversion state (if that matters). If `off` is -1, the resulting object represents an invalid stream position.

The second constructor stores a zero offset and the object `state`.

Member functions:

fpos::operator!=:

```
bool operator!=(const fpos& rhs) const;
```

The member function returns `!(*this == rhs)`.

fpos::operator+:

```
fpos operator+(streamoff off) const;
```

The member function returns `fpos(*this) += off`.

fpos::operator+=:

```
fpos& operator+=(streamoff off);
```

The member function adds `off` to the stored offset member object, then returns `*this`. For positioning within a file, the result is generally valid only for binary streams that do not have a state-dependent encoding.

fpos::operator-:

```
streamoff operator-(const fpos& rhs) const;
fpos operator-(streamoff off) const;
```

The first member function returns `(streamoff)*this - (streamoff)rhs`. The second member function returns `fpos(*this) -= off`.

fpos::operator-=:

```
fpos& operator-=(streamoff off);
```

The member function returns `fpos(*this) -= off`. For positioning within a file, the result is generally valid only for binary streams that do not have a state-dependent encoding.

fpos::operator==:

```
bool operator==(const fpos& rhs) const;
```

The member function returns `(streamoff)*this == (streamoff)rhs`.

fpos::operator streamoff:

```
operator streamoff() const;
```

The member function returns the stored offset member object, plus any additional offset stored as part of the `fpos_t` member object.

fpos::state:

```
St state() const;
void state(St state);
```

The first member function returns the value stored in the `St` member object. The second member function stores `state` in the `St` member object.

ios_base

Description: The class describes the storage and member functions common to both input and output streams that does not depend on the template parameters. (The template class basic_ios describes what is common and *is* dependent on template parameters.)

An object of class `ios_base` stores **formatting information**, which consists of:

- **format flags** in an object of type fmtflags
- an **exception mask** in an object of type iostate
- a **field width** in an object of type *int*
- a **display precison** in an object of type *int*
- a **locale object** in an object of type locale
- two **extensible arrays**, with elements of type *long* and *void* pointer

An object of class `ios_base` also stores **stream state information**, in an object of type iostate, and a **callback stack**.

Synopsis:

```
class ios_base {
public:
    class failure;
    typedef T1 fmtflags;
    static const fmtflags boolalpha, dec, fixed, hex,
        internal, left, oct, right, scientific,
        showbase, showpoint, showpos, skipws, unitbuf,
        uppercase, adjustfield, basefield, floatfield;
    typedef T2 iostate;
    static const iostate badbit, eofbit, failbit,
        goodbit;
    typedef T3 openmode;
    static const openmode app, ate, binary, in, out,
        trunc;
    typedef T4 seekdir;
    static const seekdir beg, cur, end;
    typedef T5 event;
    static const event copyfmt_event, erase_event,
        copyfmt_event;
    class Init;
    fmtflags flags() const;
    fmtflags flags(fmtflags fmtfl);
```

```
    fmtflags setf(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl, fmtflags mask);
    void unsetf(fmtflags mask);
    streamsize precision() const;
    streamsize precision(streamsize prec);
    streamsize width() const;
    stramsize width(streamsize wide);
    locale imbue(const locale& loc);
    locale getloc() const;
    static int xalloc();
    long& iword(int idx);
    void *& pword(int idx);
    typedef void *(event_callback(event ev,
        ios_base& ios, int idx);
    void register_callback(event_callback pfn, int idx);
    static bool sync_with_stdio(bool sync = true);
protected:
    ios_base();
private:
    ios_base(const ios_base&);
    ios_base& operator=(const ios_base&);
    };
```

Constructor:

ios_base::ios_base:

```
ios_base();
```

The (protected) constructor does nothing. A later call to `basic_ios::init` *must* initialize the object before it can be safely destroyed. Thus, the only safe use for class `ios_base` is as a base class for template class basic_ios.

Types:

ios_base::event:

```
typedef T5 event;
static const event copyfmt_event, erase_event,
    imbue_event;
```

The type is an enumerated type `T5` that describes an object that can store the **callback event** used as an argument to a function registered with register_callback. The distinct event values are:

- `copyfmt_event`, to identify a callback that occurs near the end of a call to `copyfmt`, just before the `exception mask` is copied.
- `erase_event`, to identify a callback that occurs at the beginning of a call to `copyfmt`, or at the beginning of a call to the destructor for `*this`.
- `imbue_event`, to identify a callback that occurs at the end of a call to imbue, just before the function returns.

ios_base::event_callback:

```
typedef void *(event_callback(event ev,
        ios_base& ios, int idx);
```

The type describes a pointer to a function that can be registered with register_callback. Such a function must not throw an exception.

ios_base::fmtflags:

```
typedef T1 fmtflags;
static const fmtflags boolalpha, dec, fixed, hex,
    internal, left, oct, right, scientific,
    showbase, showpoint, showpos, skipws, unitbuf,
    uppercase, adjustfield, basefield, floatfield;
```

The type is a bitmask type `T1` that describes an object that can store format flags. The distinct flag values (elements) are:

- `boolalpha`, to insert or extract objects of type *bool* as names (such as `true` and `false`) rather than as numeric values
- `dec`, to insert or extract integer values in decimal format
- `fixed`, to insert floating-point values in fixed-point format (with no exponent field)
- `hex`, to insert or extract integer values in hexadecimal format
- `internal`, to pad to a field width as needed by inserting fill characters at a point internal to a generated numeric field
- `left`, to pad to a field width as needed by inserting fill characters at the end of a generated field (left justification)
- `oct`, to insert or extract integer values in octal format
- `right`, to pad to a field width as needed by inserting fill characters at the beginning of a generated field (right justification)
- `scientific`, to insert floating-point values in scientific format (with an exponent field)
- `showbase`, to insert a prefix that reveals the base of a generated integer field
- `showpoint`, to insert a decimal point unconditionally in a generated floating-point field
- `showpos`, to insert a plus sign in a non-negative generated numeric field
- `skipws`, to skip leading white space before certain extractions
- `unitbuf`, to flush output after each insertion
- `uppercase`, to insert uppercase equivalents of lowercase letters in certain insertions

In addition, several useful values are:

- `adjustfield`, `internal | left | right`
- `basefield`, `dec | hex | oct`
- `floatfield`, `fixed | scientific`

ios_base::iostate:

```
typedef T2 iostate;
static const iostate badbit, eofbit, failbit, goodbit;
```

The type is a bitmask type `T2` that describes an object that can store stream state information. The distinct flag values elements are:

- `badbit`, to record a loss of integrity of the stream buffer
- `eofbit`, to record end-of-file while extracting from a stream
- `failbit`, to record a failure to extract a valid field from a stream

In addition, a useful value is:

- `goodbit`, no bits set

ios_base::openmode:

```
typedef T3 openmode;
static const openmode app, ate, binary, in, out, trunc;
```

The type is a bitmask type T3 that describes an object that can store the **opening mode** for several iostreams objects. The distinct flag values (elements) are:

- app, to seek to the end of a stream before each insertion
- ate, to seek to the end of a stream when its controlling object is first created
- binary, to read a file as a binary stream, rather than as a text stream
- in, to permit extraction from a stream
- out, to permit insertion to a stream
- trunc, to truncate an existing file when its controlling object is first created

ios_base::seekdir:

```
typedef T4 seekdir;
static const seekdir beg, cur, end;
```

The type is an enumerated type T4 that describes an object that can store the **seek mode** used as an argument to the member functions of several iostreams classes. The distinct flag values are:

- beg, to seek (alter the current read or write position) relative to the beginning oc a sequence (array, stream, or file)
- cur, to seek relative to the current position within a sequence
- end, to seek relative to the end of a sequence

Member classes:

ios_base::failure:

```
class failure : public exception {
public:
    explicit failure(const string& what_arg) {
    };
```

The member class serves as the base class for all exceptions thrown by the member function clear in template class basic_ios. The value returned by what() is what_arg.data().

ios_base::Init:

```
class Init {
    };
```

The nested class describes an object whose construction ensures that the standard iostreams objects are properly constructed, even before the execution of a constructor for an arbitrary static object.

Member functions:

ios_base::flags:

```
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
```

The first member function returns the stored format flags. The second member function stores fmtfl in the format flags and returns its previous stored value.

ios_base::getloc:

```
locale getloc() const;
```

The member function returns the stored locale object.

ios_base::imbue:

```
locale imbue(const locale& loc);
```

The member function stores loc in the locale object, then reports the callback event imbue_event. It returns the previous stored value.

ios_base::iword:

```
long& iword(int idx);
```

The member function returns a reference to element idx of the extensible array with elements of type *long*. All elements are effectively present and initially store the value zero. The returned reference is invalid after the next call to iword for the object, after the object is altered by a call to basic_ios::copyfmt, or after the object is destroyed.

If idx is negative, or if unique storage is unavailable for the element, the function calls setstate(badbit) and returns a reference that might not be unique.

To obtain a unique index, for use across all objects of type ios_base, call xalloc.

ios_base::precision:

```
streamsize precision() const;
streamsize precision(streamsize prec);
```

The first member function returns the stored display precision. The second member function stores prec in the display precision and returns its previous stored value.

ios_base::pword:

```
void *& pword(int idx);
```

The member function returns a reference to element idx of the extensible array with elements of type *void* pointer. All elements are effectively present and initially store the null pointer. The returned reference is invalid after the next call to pword for the object, after the object is altered by a call to basic_ios::copyfmt, or after the object is destroyed.

If idx is negative, or if unique storage is unavailable for the element, the function calls setstate(badbit) and returns a reference that might not be unique.

To obtain a unique index, for use across all objects of type ios_base, call xalloc.

ios_base::register_callback:

```
void register_callback(event_callback pfn, int idx);
```

The member function pushes the pair {pfn, idx} onto the stored callback stack. When a callback event ev is reported, the functions are called, in reverse order of registry, by the expression (*pfn)(ev, *this, idx).

ios_base::setf:

```
void setf(fmtflags mask);
fmtflags setf(fmtflags fmtfl, fmtflags mask);
```

The first member function effectively calls `flags(mask | flags())` (set selected bits), then returns the previous format flags. The second member function effectively calls `flags(mask & fmtfl, flags() & ~mask)` (replace selected bits under a mask), then returns the previous format flags.

ios_base::sync_with_stdio:

```
static bool sync_with_stdio(bool sync = true);
```

The static member function stores a **stdio sync flag**, which is initially true. When true, this flag ensures that operations on the same file are properly synchronized between the iostreams functions and those defined in the Standard C library. Otherwise, synchronization may or may not be guaranteed, but performance may be improved. The function stores `sync` in the stdio sync flag and returns its previous stored value. You can call it reliably only before performing any operations on the standard streams.

ios_base::unsetf:

```
void unsetf(fmtflags mask);
```

The member function effectively calls `flags(~mask & flags())` (clear selected bits).

ios_base::width:

```
streamsize width() const;
streamsize width(streamsize wide);
```

The first member function returns the stored field width. The second member function stores `wide` in the field width and returns its previous stored value.

ios_base::xalloc:

```
static int xalloc();
```

The static member function returns a stored static value, which it increments on each call. You can use the return value as a unique index argument when calling the member functions iword or pword.

Manipulators

boolalpha

```
ios_base& boolalpha(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: boolalpha)`, then returns `str`.

noboolalpha

```
ios_base& noboolalpha(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::boolalpha)`, then returns `str`.

showbase

```
ios_base& showbase(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::showbase)`, then returns `str`.

noshowbase

```
ios_base& noshowbase(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::showbase)`, then returns `str`.

showpoint

```
ios_base& showpoint(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::showpoint)`, then returns `str`.

noshowpoint

```
ios_base& noshowpoint(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::showpoint)`, then returns `str`.

showpos

```
ios_base& showpos(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::showpos)`, then returns `str`.

noshowpos

```
ios_base& noshowpos(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::showpos")`, then returns `str`.

skipws

```
ios_base& skipws(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::skipws)`, then returns `str`.

noskipws

```
ios_base& noskipws(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::skipws)`, then returns `str`.

unitbuf

```
ios_base& unitbuf(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::unitbuf)`, then returns `str`.

nounitbuf

```
ios_base& nounitbuf(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::unitbuf)`, then returns `str`.

uppercase

```
ios_base& uppercase(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::uppercase)`, then returns `str`.

nouppercase

```
ios_base& nouppercase(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base::uppercase)`, then returns `str`.

internal

```
ios_base& internal(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: internal, ios_base:: adjustfield)`, then returns `str`.

left

```
ios_base& left(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: left, ios_base:: adjustfield)`, then returns `str`.

right

```
ios_base& right(ios_base& str);
```

The maiipulator effectively calls `str.setf(ios_base::right, ios_base::adjustfield)`, then returns `str`.

dec

```
ios_base& dec(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: dec, ios_base:: basefield)`, then returns `str`.

hex

```
ios_base& hex(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: hex, ios_base:: basefield)`, then returns `str`.

hexfloat

```
namespace std {
namespace tr1 {
    ios_base& hexfloat(ios_base& str);
} // namespace tr1
} // namespace std
```

The manipulator behaves as though it calls `str.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield)` , then returns `str`.

CAVEAT:
: Although C++2003 gives no meaning to the combination of `ios_base::fixed` and `ios_base::scientific`, this TR1 implementation already has a well-defined behavior when these two formatting flags are combined. As such, although this TR1 implementation recognizes the `hexfloat` manipulator, it still uses the old semantics in the formatted output, to maintain backwards compatibility. In other words, the call to `hexfloat` is ignored, and the conversion specifiers %a and %A are not used.

Note: To enable the TR1 headers, you must define the macro **__IBMCPP_TR1__** as 1.

oct

```
ios_base& oct(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::oct, ios_base::basefield)`, then returns `str`.

fixed

```
ios_base& fixed(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: fixed, ios_base:: floatfield)`, then returns `str`.

scientific

```
ios_base& scientific(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::scientific, ios_base::floatfield)`, then returns `str`.

Types

ios

```
typedef basic_ios<char, char_traits<char> > ios;
```

The type is a synonym for template class basic_ios, specialized for elements of type *char* with default character traits.

streamoff

```
typedef T1 streamoff;
```

The type is a signed integer type `T1` that describes an object that can store a byte offset involved in various stream positioning operations. Its representation has at least 32 value bits. It is *not* necessarily large enough to represent an arbitrary byte position within a stream. The value `streamoff(-1)` generally indicates an erroneous offset.

streampos

```
typedef fpos<mbstate_t> streampos;
```

The type is a synonym for `fpos< mbstate_t>`.

streamsize

```
typedef T2 streamsize;
```

The type is a signed integer type `T3` that describes an object that can store a count of the number of elements involved in various stream operations. Its representation has at least 16 bits. It is *not* necessarily large enough to represent an arbitrary byte position within a stream.

wios

```
typedef basic_ios<wchar_t, char_traits<wchar_t> > wios;
```

The type is a synonym for template class basic_ios, specialized for elements of type `wchar_t` with default character traits.

wstreampos

```
typedef fpos<mbstate_t> wstreampos;
```

The type is a synonym for `fpos< wmbstate_t>`.

<iosfwd>

Description

Include the iostreams standard header **<iosfwd>** to declare forward references to several template classes used throughout iostreams. All such template classes are defined in other standard headers. You include this header explicitly only when you need one of the above declarations, but not its definition.

Synopsis

```
namespace std {
typedef T1 streamoff;
typedef T2 streamsize;
typedef fpos streampos;

       // TEMPLATE CLASSES
template<class E>
    class char_traits;
class char_traits<char>;
class char_traits<wchar_t>;
template<class E, class T = char_traits<E> >
    class basic_ios;
template<class E, class T = char_traits<E> >
    class istreambuf_iterator;
template<class E, class T = char_traits<E> >
    class ostreambuf_iterator;
template<class E, class T = char_traits<E> >
    class basic_streambuf;
template<class E, class T = char_traits<E> >
    class basic_istream;
template<class E, class T = char_traits<E> >
    class basic_ostream;
template<class E, class T = char_traits<E> >
    class basic_iostream;
template<class E, class T = char_traits<E> >
    class basic_stringbuf;
template<class E, class T = char_traits<E> >
    class basic_istringstream;
template<class E, class T = char_traits<E> >
    class basic_ostringstream;
template<class E, class T = char_traits<E> >
    class basic_stringstream;
template<class E, class T = char_traits<E> >
    class basic_filebuf;
template<class E, class T = char_traits<E> >
    class basic_ifstream;
template<class E, class T = char_traits<E> >
    class basic_ofstream;
template<class E, class T = char_traits<E> >
    class basic_fstream;

        // char TYPE DEFINITIONS
typedef basic_ios<char, char_traits<char> > ios;
typedef basic_streambuf<char, char_traits<char> >
    streambuf;
typedef basic_istream<char, char_traits<char> >
    istream;
typedef basic_ostream<char, char_traits<char> >
    ostream;
typedef basic_iostream<char, char_traits<char> >
    iostream;
typedef basic_stringbuf<char, char_traits<char> >
    stringbuf;
typedef basic_istringstream<char, char_traits<char> >
```

```
    istringstream;
typedef basic_ostringstream<char, char_traits<char> >
    ostringstream;
typedef basic_stringstream<char, char_traits<char> >
    stringstream;
typedef basic_filebuf<char, char_traits<char> >
    filebuf;
typedef basic_ifstream<char, char_traits<char> >
    ifstream;
typedef basic_ofstream<char, char_traits<char> >
    ofstream;
typedef basic_fstream<char, char_traits<char> >
    fstream;

        // wchar_t TYPE DEFINITIONS
typedef basic_ios<wchar_t, char_traits<wchar_t> > wios;
typedef basic_streambuf<wchar_t, char_traits<wchar_t> >
    wstreambuf;
typedef basic_istream<wchar_t, char_traits<wchar_t> >
    wistream;
typedef basic_ostream<wchar_t, char_traits<wchar_t> >
    wostream;
typedef basic_iostream<wchar_t, char_traits<wchar_t> >
    wiostream;
typedef basic_stringbuf<wchar_t, char_traits<wchar_t> >
    wstringbuf;
typedef basic_istringstream<wchar_t,
    char_traits<wchar_t> > wistringstream;
typedef basic_ostringstream<wchar_t,
    char_traits<wchar_t> > wostringstream;
typedef basic_stringstream<wchar_t,
    char_traits<wchar_t> > wstringstream;
typedef basic_filebuf<wchar_t, char_traits<wchar_t> >
    wfilebuf;
typedef basic_ifstream<wchar_t, char_traits<wchar_t> >
    wifstream;
typedef basic_ofstream<wchar_t, char_traits<wchar_t> >
    wofstream;
typedef basic_fstream<wchar_t, char_traits<wchar_t> >
    wfstream;
    }
```

<iostream>

Description

Include the iostreams standard header **<iostream>** to declare objects that control reading from and writing to the standard streams. This is often the *only* header you need include to perform input and output from a C++ program.

The objects fall into two groups:

- cin, cout, cerr, and clog are byte oriented, performing conventional byte-at-a-time transfers
- wcin, wcout, wcerr“wcerr” on page 125, and wclog are wide oriented, translating to and from the wide characters that the program manipulates internally

Once you perform certain operations on a stream, such as the standard input, you cannot perform operations of a different orientation on the same stream. Hence, a program cannot operate interchangeably on both `cin` and `wcin`, for example.

All the objects declared in this header share a peculiar property — you can assume they are **constructed** before any static objects you define, in a translation unit that includes `<iostreams>`. Equally, you can assume that these objects are **not destroyed** before the destructors for any such static objects you define. (The output streams are, however, flushed during program termination.) Hence, you can safely read from or write to the standard streams prior to program startup and after program termination.

This guarantee is *not* universal, however. A static constructor may call a function in another translation unit. The called function cannot assume that the objects declared in this header have been constructed, given the uncertain order in which translation units participate in static construction. To use these objects in such a context, you must first construct an object of class **ios_base::Init**, as in:

```
#include <iostream>
void marker()
    {    // called by some constructor
    ios_base::Init unused_name;
    cout << "called fun" << endl;
    }
```

Synopsis

```
namespace std {
extern istream cin;
extern ostream cout;
extern ostream cerr;
extern ostream clog;

extern wistream wcin;
extern wostream wcout;
extern wostream wcerr;
extern wostream wclog;
    }
```

Objects

cerr

```
extern ostream cerr;
```

The object controls unbuffered insertions to the standard error output as a byte stream. Once the object is constructed, the expression `cerr.flags() & unitbuf` is nonzero.

cin

```
extern istream cin;
```

The object controls extractions from the standard input as a byte stream. Once the object is constructed, the call `cin.tie()` returns `&cout`.

clog

```
extern ostream clog;
```

The object controls buffered insertions to the standard error output as a byte stream.

cout

```
extern ostream cout;
```

The object controls insertions to the standard output as a byte stream.

wcerr

```
extern wostream wcerr;
```

The object controls unbuffered insertions to the standard error output as a wide stream. Once the object is constructed, the expression `wcerr.flags() & unitbuf` is nonzero.

wcin

```
extern wistream wcin;
```

The object controls extractions from the standard input as a wide stream. Once the object is constructed, the call `wcin.tie()` returns `&wcout`.

wclog

```
extern wostream wclog;
```

The object controls buffered insertions to the standard error output as a wide stream.

wcout

```
extern wostream wcout;
```

The object controls insertions to the standard output as a wide stream.

<istream>

Description

Include the iostreams standard header **<istream>** to define template class basic_istream, which mediates extractions for the iostreams, and the template class. basic_iostream, which mediates both insertions and extractions. The header also defines a related manipulator. (This header is typically included for you by another of the iostreams headers. You seldom have occasion to include it directly.)

Synopsis

```
namespace std {
template<class E, class T = char_traits<E> >
    class basic_istream;
typedef basic_istream<char, char_traits<char> >
    istream;
typedef basic_istream<wchar_t, char_traits<wchar_t> >
    wistream;
template<class E, class T = char_traits<E> >
    class basic_iostream;
typedef basic_iostream<char, char_traits<char> >
    iostream;
typedef basic_iostream<wchar_t, char_traits<wchar_t> >
    wiostream;

        // EXTRACTORS
template<class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is, E *s);
template<class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is, E& c);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
```

```
            signed char *s);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            signed char& c);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            unsigned char *s);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            unsigned char& c);

        // MANIPULATORS
template class<E, T>
    basic_istream<E, T>& ws(basic_istream<E, T>& is);
    }
```

Classes

basic_iostream

Description: The template class describes an object that controls insertions, through its base object `basic_ostream<E, T>`, and extractions, through its base object `basic_istream<E, T>`. The two objects share a common virtual base object `basic_ios<E, T>`. They also manage a common stream buffer, with elements of type `E`, whose character traits are determined by the class `T`. The constructor initializes its base objects via `basic_istream(sb)` and `basic_ostream(sb)`.

Synopsis:

```
template <class E, class T = char_traits<E> >
    class basic_iostream : public basic_istream<E, T>,
        public basic_ostream<E, T> {
public:
    explicit basic_iostream(basic_streambuf<E, T>& *sb);
    virtual ~basic_iostream();
    };
```

basic_istream

Description: The template class describes an object that controls extraction of elements and encoded objects from a stream buffer with elements of type `E`, also known as char_type, whose character traits are determined by the class `T`, also known as traits_type.

Most of the member functions that overload operator>> are **formatted input functions**. They follow the pattern:

```
    iostate state = goodbit;
    const sentry ok(*this);
    if (ok)
        {try
            {<extract elements and convert
            accumulate flags in state
            store a successful conversion>}
        catch (...)
            {try
                {setstate(badbit); }
   catch (...)
    {}
```

```
        if ((exceptions() & badbit) != 0)
            throw; }}
setstate(state);
return (*this);
```

Many other member functions are **unformatted input functions**. They follow the pattern:

```
iostate state = goodbit;
count = 0;    // the value returned by gcount
const sentry ok(*this, true);
if (ok)
    {try
        {<extract elements and deliver
        count extracted elements in count
        accumulate flags in state>}
    catch (...)
        {try
            {setstate(badbit); }
catch (...)
{}
        if ((exceptions() & badbit) != 0)
            throw; }}
setstate(state);
```

Both groups of functions call `setstate(eofbit)` if they encounter end-of-file while extracting elements.

An object of class `basic_istream<E, T>` stores:

- a virtual public base object of class `basic_ios<E, T>`
- an **extraction count** for the last unformatted input operation (called `count` in the code above

Synopsis:

```
template <class E, class T = char_traits<E> >
    class basic_istream
        : virtual public basic_ios<E, T> {
public:
    typedef typename basic_ios<E, T>::char_type char_type;
    typedef typename basic_ios<E, T>::traits_type traits_type;
    typedef typename basic_ios<E, T>::int_type int_type;
    typedef typename basic_ios<E, T>::pos_type pos_type;
    typedef typename basic_ios<E, T>::off_type off_type;
    explicit basic_istream(basic_streambuf<E, T> *sb);
    class sentry;
    virtual ~istream();
    bool ipfx(bool noskip = false);
    void isfx();
    basic_istream& operator>>(
        basic_istream& (*pf)(basic_istream&));
    basic_istream& operator>>(
        ios_base& (*pf)(ios_base&));
    basic_istream& operator>>(
        basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
    basic_istream& operator>>(
        basic_streambuf<E, T> *sb);
    basic_istream& operator>>(bool& n);
    basic_istream& operator>>(short& n);
    basic_istream& operator>>(unsigned short& n);
    basic_istream& operator>>(int& n);
    basic_istream& operator>>(unsigned int& n);
    basic_istream& operator>>(long& n);
    basic_istream& operator>>(unsigned long& n);
    basic_istream& operator>>(void *& n);
```

```
    basic_istream& operator>>(float& n);
    basic_istream& operator>>(double& n);
    basic_istream& operator>>(long double& n);
    streamsize gcount() const;
    int_type get();
    basic_istream& get(char_type& c);
    basic_istream& get(char_type *s, streamsize n);
    basic_istream&
        get(char_type *s, streamsize n, char_type delim);
    basic_istream&
        get(basic_streambuf<char_type, T> *sb);
    basic_istream&
        get(basic_streambuf<E, T> *sb, char_type delim);
    basic_istream& getline(char_type *s, streamsize n);
    basic_istream& getline(char_type *s, streamsize n,
        char_type delim);
    basic_istream& ignore(streamsize n = 1,
        int_type delim = traits_type::eof());
    int_type peek();
    basic_istream& read(char_type *s, streamsize n);
    streamsize readsome(char_type *s, streamsize n);
    basic_istream& putback(char_type c);
    basic_istream& unget();
    pos_type tellg();
    basic_istream& seekg(pos_type pos);
    basic_istream& seekg(off_type off,
        ios_base::seek_dir way);
    int sync();
    };
```

Constructor:

basic_istream::basic_istream:

```
explicit basic_istream(basic_streambuf<E, T> *sb);
```

The constructor initializes the base class by calling `init(sb)`. It also stores zero in the extraction count.

Member classes:

basic_istream::sentry:

```
class sentry {
public:
    explicit sentry(basic_istream& is,
        bool noskip = false);
    operator bool() const;
    };
```

The nested class describes an object whose declaration structures the formatted input functions and the unformatted input functions. The constructor effectively calls `is.ipfx(noskip)` and stores the return value. `operator bool()` delivers this return value. The destructor effectively calls `is.isfx()`.

Member functions:

basic_istream::gcount:

```
streamsize gcount() const;
```

The member function returns the extraction count.

basic_istream::ipfx:

```
bool ipfx(bool noskip = false);
```

The member function prepares for formatted or unformatted input. If `good()` is true, the function:

- calls tie-> `flush()` if `tie()` is not a null pointer
- effectively calls `ws(*this)` if `flags() & skipws` is nonzero

If, after any such preparation, `good()` is false, the function calls `setstate(failbit)`. In any case, the function returns `good()`.

You should not call `ipfx` directly. It is called as needed by an object of class sentry.

basic_istream::isfx:

```
void isfx();
```

The member function has no official duties, but an implementation may depend on a call to `isfx` by a formatted or unformatted input function to tidy up after an extraction. You should not call `isfx` directly. It is called as needed by an object of class sentry.

basic_istream::operator>>:

```
basic_istream& operator>>(
    basic_istream& (*pf)(basic_istream&));
basic_istream& operator>>(
    ios_base& (*pf)(ios_base&));
basic_istream& operator>>(
    basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
basic_istream& operator>>(
    basic_streambuf<E, T> *sb);
basic_istream& operator>>(bool& n);
basic_istream& operator>>(short& n);
basic_istream& operator>>(unsigned short& n);
basic_istream& operator>>(int& n);
basic_istream& operator>>(unsigned int& n);
basic_istream& operator>>(long& n);
basic_istream& operator>>(unsigned long& n);
basic_istream& operator>>(void *& n);
basic_istream& operator>>(float& n);
basic_istream& operator>>(double& n);
basic_istream& operator>>(long double& n);
```

The first member function ensures that an expression of the form `istr >> ws` calls `ws(istr)`, then returns `*this`. The second and third functions ensure that other manipulators, such as hex behave similarly. The remaining functions constitute the formatted input functions.

The function:

```
basic_istream& operator>>(
    basic_streambuf<E, T> *sb);
```

extracts elements, if `sb` is not a null pointer, and inserts them in `sb`. Extraction stops on end-of-file. It also stops, without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls `setstate(failbit)`. In any case, the function returns `*this`.

The function:

```
basic_istream& operator>>(bool& n);
```

extracts a field and converts it to a boolean value by calling use_facet<num_get<E, InIt>(getloc()). get(InIt(rdbuf()), Init(0), *this, getloc(), n). Here, InIt is defined as istreambuf_iterator<E, T>. The function returns *this.

The functions:

```
basic_istream& operator>>(short& n);
basic_istream& operator>>(unsigned short& n);
basic_istream& operator>>(int& n);
basic_istream& operator>>(unsigned int& n);
basic_istream& operator>>(long& n);
basic_istream& operator>>(unsigned long& n);
basic_istream& operator>>(void *& n);
```

each extract a field and convert it to a numeric value by calling use_facet<num_get<E, InIt>(getloc()). get(InIt(rdbuf()), Init(0), *this, getloc(), x). Here, InIt is defined as istreambuf_iterator<E, T>, and x has type *long, unsigned long,* or *void* * as needed.

If the converted value cannot be represented as the type of n, the function calls setstate(failbit). In any case, the function returns *this.

The functions:

```
basic_istream& operator>>(float& n);
basic_istream& operator>>(double& n);
basic_istream& operator>>(long double& n);
```

each extract a field and convert it to a numeric value by calling use_facet<num_get<E, InIt>(getloc()). get(InIt(rdbuf()), Init(0), *this, getloc(), x). Here, InIt is defined as istreambuf_iterator<E, T>, and x has type *double* or *long double* as needed.

If the converted value cannot be represented as the type of n, the function calls setstate(failbit). In any case, it returns *this.

basic_istream::readsome:

```
streamsize readsome(char_type *s, streamsize n);
```

The member function extracts up to n elements and stores them in the array beginning at s. If rdbuf() is a null pointer, the function calls setstate(failbit). Otherwise, it assigns the value of rdbuf()->in_avail() to N. if N < 0, the function calls setstate(eofbit). Otherwise, it replaces the value stored in N with the smaller of n and N, then calls read(s, N). In any case, the function returns gcount().

basic_istream::seekg:

```
basic_istream& seekg(pos_type pos);
basic_istream& seekg(off_type off,
    ios_base::seek_dir way);
```

If fail() is false, the first member function calls rdbuf()-> pubseekpos(pos). If fail() is false, the second function calls rdbuf()-> pubseekoff(off, way). Both functions return *this.

basic_istream::sync:

```
int sync();
```

If `rdbuf()` is a null pointer, the function returns -1. Otherwise, it calls `rdbuf()->pubsync()`. If that returns -1, the function calls `setstate(badbit)` and returns -1. Otherwise, the function returns zero.

basic_istream::tellg:

```
pos_type tellg();
```

If `fail()` is false, the member function returns `rdbuf()-> pubseekoff(0, cur, in)`. Otherwise, it returns `pos_type(-1)`.

basic_istream::get:

```
int_type get();
basic_istream& get(char_type& c);
basic_istream& get(char_type *s, streamsize n);
basic_istream& get(char_type *s, streamsize n,
    char_type delim);
basic_istream& get(basic_streambuf<E, T> *sb);
basic_istream& get(basic_streambuf<E, T> *sb,
    char_type delim);
```

The first of these unformatted input functions extracts an element, if possible, as if by returning `rdbuf()->sbumpc()`. Otherwise, it returns `traits_type::eof()`. If the function extracts no element, it calls `setstate(failbit)`.

The second function extracts the int_type element `x` the same way. If `x` compares equal to `traits_type::eof(x)`, the function calls `setstate(failbit)`. Otherwise, it stores `traits_type::to_char_type(x)` in `c`. The function returns `*this`.

The third function returns `get(s, n, widen('\n'))`.

The fourth function extracts up to `n - 1` elements and stores them in the array beginning at `s`. It always stores `char_type()` after any extracted elements it stores. In order of testing, extraction stops:

1. at end of file
2. after the function extracts an element that compares equal to `delim`, in which case the element is put back to the controlled sequence
3. after the function extracts `n - 1` elements

If the function extracts no elements, it calls `setstate(failbit)`. In any case, it returns `*this`.

The fifth function returns `get(sb, widen('\n'))`.

The sixth function extracts elements and inserts them in `sb`. Extraction stops on end-of-file or on an element that compares equal to `delim` (which is not extracted). It also stops, without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls `setstate(failbit)`. In any case, the function returns `*this`.

basic_istream::getline:

```
basic_istream& getline(char_type *s, streamsize n);
basic_istream& getline(char_type *s, streamsize n,
    char_type delim);
```

The first of these unformatted input functions returns `getline(s, n, widen('\n'))`.

The second function extracts up to `n - 1` elements and stores them in the array beginning at `s`. It always stores `char_type()` after any extracted elements it stores. In order of testing, extraction stops:

1. at end of file
2. after the function extracts an element that compares equal to `delim`, in which case the element is neither put back nor appended to the controlled sequence
3. after the function extracts `n - 1` elements

If the function extracts no elements or `n - 1` elements, it calls `setstate(failbit)`. In any case, it returns `*this`.

basic_istream::ignore:

```
basic_istream& ignore(streamsize n = 1,
    int_type delim = traits_type::eof());
```

The unformatted input function extracts up to `n` elements and discards them. If `n` equals `numeric_limits<int>::max()`, however, it is taken as arbitrarily large. Extraction stops early on end-of-file or on an element `x` such that `traits_type::to_int_type(x)` compares equal to `delim` (which is also extracted). The function returns `*this`.

basic_istream::peek:

```
int_type peek();
```

The unformatted input function extracts an element, if possible, as if by returning `rdbuf()->sgetc()`. Otherwise, it returns `traits_type::eof()`.

basic_istream::putback:

```
basic_istream& putback(char_type c);
```

The unformatted input function puts back `c`, if possible, as if by calling `rdbuf()->sputbackc()`. If `rdbuf()` is a null pointer, or if the call to `sputbackc` returns `traits_type::eof()`, the function calls `setstate(badbit)`. In any case, it returns `*this`.

basic_istream::read:

```
basic_istream& read(char_type *s, streamsize n);
```

The unformatted input function extracts up to `n` elements and stores them in the array beginning at `s`. Extraction stops early on end-of-file, in which case the function calls `setstate(failbit)`. In any case, it returns `*this`.

basic_istream::unget:

```
basic_istream& unget();
```

The unformatted input function puts back the previous element in the stream, if possible, as if by calling `rdbuf()->sungetc()`. If `rdbuf()` is a null pointer, or if the call to `sungetc` returns `traits_type::eof()`, the function calls `setstate(badbit)`. In any case, it returns `*this`.

Manipulators

ws

```
template class<E, T>
    basic_istream<E, T>& ws(basic_istream<E, T>& is);
```

The manipulator extracts and discards any elements `x` for which `use_facet< ctype<E> >( getloc()). is( ctype<E>::space, x)` is true.

The function calls `setstate(eofbit)` if it encounters end-of-file while extracting elements. It returns `is`.

Types

iostream

```
typedef basic_iostream<char, char_traits<char> > iostream;
```

The type is a synonym for template class basic_iostream, specialized for elements of type *char* with default character traits.

istream

```
typedef basic_istream<char, char_traits<char> > istream;
```

The type is a synonym for template class basic_istream, specialized for elements of type *char* with default character traits.

wiostream

```
typedef basic_iostream<wchar_t, char_traits<wchar_t> >
    wiostream;
```

The type is a synonym for template class basic_iostream, specialized for elements of type `wchar_t` with default character traits.

wistream

```
typedef basic_istream<wchar_t, char_traits<wchar_t> >
    wistream;
```

The type is a synonym for template class basic_istream, specialized for elements of type `wchar_t` with default character traits.

Template functions

operator>>

```
template<class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is, E *s);
template<class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is, E& c);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            signed char *s);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            signed char& c);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            unsigned char *s);
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            unsigned char& c);
```

The template function:

```
template<class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is, E *s);
```

extracts up to `n - 1` elements and stores them in the array beginning at `s`. If `is.width()` is greater than zero, `n` is `is.width()`; otherwise it is the largest array of `E` that can be declared. The function always stores `E()` after any extracted elements it stores. Extraction stops early on end-of-file or on any element (which is not extracted) that would be discarded by ws. If the function extracts no elements, it calls `is.setstate(failbit)`. In any case, it calls `is.width(0)` and returns `is`.

The template function:

```
template<class E, class T>
    basic_istream<E, T>&
        operator>>(basic_istream<E, T>& is, char& c);
```

extracts an element, if possible, and stores it in `c`. Otherwise, it calls `is.setstate(failbit)`. In any case, it returns `is`.

The template function:

```
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            signed char *s);
```

returns `is >> (char *)s`.

The template function:

```
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            signed char& c);
```

returns `is >> (char&)c`.

The template function:

```
template<class T>
    basic_istream<char, T>&
       operator>>(basic_istream<char, T>& is,
           unsigned char *s);
```

returns `is >> (char *)s`.

The template function:

```
template<class T>
    basic_istream<char, T>&
        operator>>(basic_istream<char, T>& is,
            unsigned char& c);
```

returns `is >> (char&)c`.

<iterator>

Description

Include the STL standard header **<iterator>** to define a number of classes, template classes, and template functions that aid in the declaration and manipulation of iterators.

Synopsis

```
namespace std {
struct input_iterator_tag;
struct output_iterator_tag;
struct forward_iterator_tag;
struct bidirectional_iterator_tag;
struct random_access_iterator_tag;

        // TEMPLATE CLASSES
template<class C, class T, class Dist,
    class Pt, class Rt>
    struct iterator;
template<class It>
    struct iterator_traits;
template<class T>
    struct iterator_traits<T *>
template<class RanIt>
    class reverse_iterator;
template<class Cont>
    class back_insert_iterator;
template<class Cont>
    class front_insert_iterator;
template<class Cont>
    class insert_iterator;
template<class U, class E, class T, class Dist>
    class istream_iterator;
template<class U, class E, class T>
    class ostream_iterator;
template<class E, class T>
    class istreambuf_iterator;
template<class E, class T>
    class ostreambuf_iterator;

        // TEMPLATE FUNCTIONS
template<class RanIt>
    bool operator==(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class U, class E, class T, class Dist>
    bool operator==(
        const istream_iterator<U, E, T, Dist>& lhs,
        const istream_iterator<U, E, T, Dist>& rhs);
template<class E, class T>
    bool operator==(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
template<class RanIt>
    bool operator!=(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class U, class E, class T, class Dist>
    bool operator!=(
        const istream_iterator<U, E, T, Dist>& lhs,
        const istream_iterator<U, E, T, Dist>& rhs);
template<class E, class T>
    bool operator!=(
```

```
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
template<class RanIt>
    bool operator<(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    bool operator>(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    bool operator<=(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    bool operator>=(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    Dist operator-(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class RanIt>
    reverse_iterator<RanIt> operator+(
        Dist n,
        const reverse_iterator<RanIt>& rhs);
template<class Cont>
    back_insert_iterator<Cont> back_inserter(Cont& x);
template<class Cont>
    front_insert_iterator<Cont> front_inserter(Cont& x);
template<class Cont, class Iter>
    insert_iterator<Cont> inserter(Cont& x, Iter it);
template<class InIt, class Dist>
    void advance(InIt& it, Dist n);
template<class Init, class Dist>
    iterator_traits<InIt>::difference_type
        distance(InIt first, InIt last);
    }
```

Classes

back_insert_iterator

Description: The template class describes an output iterator object. It inserts elements into a container of type **Cont**, which it accesses via the protected pointer object it stores called **container**. The container must define:

- the member type **const_reference**, which is the type of a constant reference to an element of the sequence controlled by the container
- the member type **reference**, which is the type of a reference to an element of the sequence controlled by the container
- the member type **value_type**, which is the type of an element of the sequence controlled by the container
- the member function **push_back(value_type c)**, which appends a new element with value `c` to the end of the sequence

Synopsis:

```
template<class Cont>
    class back_insert_iterator
        : public iterator<output_iterator_tag,
            void, void, void, void> {
public:
    typedef Cont container_type;
```

```
    typedef typename Cont::reference reference;
    typedef typename Cont::value_type value_type;
    explicit back_insert_iterator(Cont& x);
    back_insert_iterator&
        operator=(typename Cont::const_reference val);
    back_insert_iterator& operator*();
    back_insert_iterator& operator++();
    back_insert_iterator operator++(int);
protected:
    Cont *container;
    };
```

Constructor:

back_insert_iterator::back_insert_iterator:

```
explicit back_insert_iterator(Cont& x);
```

The constructor initializes container with `&x`.

Types:

back_insert_iterator::container_type:

```
typedef Cont container_type;
```

The type is a synonym for the template parameter `Cont`.

back_insert_iterator::reference:

```
typedef typename Cont::reference reference;
```

The type describes a reference to an element of the sequence controlled by the associated container.

back_insert_iterator::value_type:

```
typedef typename Cont::value_type value_type;
```

The type describes the elements of the sequence controlled by the associated container.

Member functions:

back_insert_iterator::operator:*

```
back_insert_iterator& operator*();
```

The member function returns `*this`.

back_insert_iterator::operator++:

```
back_insert_iterator& operator++();
back_insert_iterator operator++(int);
```

The member functions both return `*this`.

back_insert_iterator::operator=:

```
back_insert_iterator&
    operator=(typename Cont::const_reference val);
```

The member function evaluates `container. push_back(val)`, then returns `*this`.

front_insert_iterator

Description: The template class describes an output iterator object. It inserts elements into a container of type **Cont**, which it accesses via the protected pointer object it stores called **container**. The container must define:

- the member type **const_reference**, which is the type of a constant reference to an element of the sequence controlled by the container
- the member type **reference**, which is the type of a reference to an element of the sequence controlled by the container
- the member type **value_type**, which is the type of an element of the sequence controlled by the container
- the member function **push_front(value_type c)**, which prepends a new element with value `c` to the beginning of the sequence

Synopsis:

```
template<class Cont>
    class front_insert_iterator
        : public iterator<output_iterator_tag,
            void, void, void, void> {
public:
    typedef Cont container_type;
    typedef typename Cont::reference reference;
    typedef typename Cont::value_type value_type;
    explicit front_insert_iterator(Cont& x);
    front_insert_iterator&
        operator=(typename Cont::const_reference val);
    front_insert_iterator& operator*();
    front_insert_iterator& operator++();
    front_insert_iterator operator++(int);
protected:
    Cont *container;
    };
```

Constructor:

front_insert_iterator::front_insert_iterator:

```
explicit front_insert_iterator(Cont& x);
```

The constructor initializes container with `&x`.

Types:

front_insert_iterator::container_type:

```
typedef Cont container_type;
```

The type is a synonym for the template parameter `Cont`.

front_insert_iterator::reference:

```
typedef typename Cont::reference reference;
```

The type describes a reference to an element of the sequence controlled by the associated container.

front_insert_iterator::value_type:

```
typedef typename Cont::value_type value_type;
```

The type describes the elements of the sequence controlled by the associated container.

Member functions:

front_insert_iterator::operator:*

```
front_insert_iterator& operator*();
```

The member function returns `*this`.

front_insert_iterator::operator++:

```
front_insert_iterator& operator++();
front_insert_iterator operator++(int);
```

The member functions both return `*this`.

front_insert_iterator::operator=:

```
front_insert_iterator&
    operator=(typename Cont::const_reference val);
```

The member function evaluates `container. push_front(val)`, then returns `*this`.

insert_iterator

Description: The template class describes an output iterator object. It inserts elements into a container of type **Cont**, which it accesses via the protected pointer object it stores called **container**. It also stores the protected iterator object, of class `Cont::iterator`, called **iter**. The container must define:

- the member type **const_reference**, which is the type of a constant reference to an element of the sequence controlled by the container
- the member type **iterator**, which is the type of an iterator for the container
- the member type **reference**, which is the type of a reference to an element of the sequence controlled by the container
- the member type **value_type**, which is the type of an element of the sequence controlled by the container
- the member function **insert(iterator it, value_type c)**, which inserts a new element with value `c` immediately before the element designated by `it` in the controlled sequence, then returns an iterator that designates the inserted element

Synopsis:

```
template<class Cont>
    class insert_iterator
        : public iterator<output_iterator_tag,
            void, void, void, void> {
public:
    typedef Cont container_type;
    typedef typename Cont::reference reference;
    typedef typename Cont::value_type value_type;
    insert_iterator(Cont& x,
        typename Cont::iterator it);
    insert_iterator&
        operator=(typename Cont::const_reference val);
    insert_iterator& operator*();
    insert_iterator& operator++();
    insert_iterator& operator++(int);
```

```
protected:
    Cont *container;
    typename Cont::iterator iter;
    };
```

Constructor:

insert_iterator::insert_iterator:

```
insert_iterator(Cont& x,
    typename Cont::iterator it);
```

The constructor initializes container with `&x`, and iter with `it`.

Types:

insert_iterator::container_type:

```
typedef Cont container_type;
```

The type is a synonym for the template parameter `Cont`.

insert_iterator::reference:

```
typedef typename Cont::reference reference;
```

The type describes a reference to an element of the sequence controlled by the associated container.

insert_iterator::value_type:

```
typedef typename Cont::value_type value_type;
```

The type describes the elements of the sequence controlled by the associated container.

Member functions:

insert_iterator::operator:*

```
insert_iterator& operator*();
```

The member function returns `*this`.

insert_iterator::operator++:

```
insert_iterator& operator++();
insert_iterator& operator++(int);
```

The member functions both return `*this`.

insert_iterator::operator=:

```
insert_iterator&
    operator=(typename Cont::const_reference val);
```

The member function evaluates `iter = container. insert(iter, val)`, then returns `*this`.

istream_iterator

Description: The template class describes an input iterator object. It extracts objects of class **U** from an **input stream**, which it accesses via an object it stores, of type pointer to `basic_istream<E, T>`. After constructing or incrementing an object of class `istream_iterator` with a non-null stored pointer, the object attempts to extract and store an object of type `U` from the associated input stream. If the extraction fails, the object effectively replaces the stored pointer with a null pointer (thus making an end-of-sequence indicator).

Synopsis:

```
template<class U, class E = char,
    class T = char_traits>
    class Dist = ptrdiff_t>
    class istream_iterator
        : public iterator<input_iterator_tag,
            U, Dist, U *, U&> {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef basic_istream<E, T> istream_type;
    istream_iterator();
    istream_iterator(istream_type& is);
    const U& operator*() const;
    const U *operator->() const;
    istream_iterator<U, E, T, Dist>& operator++();
    istream_iterator<U, E, T, Dist> operator++(int);
    };
```

Constructor:

istream_iterator::istream_iterator:

```
istream_iterator();
istream_iterator(istream_type& is);
```

The first constructor initializes the input stream pointer with a null pointer. The second constructor initializes the input stream pointer with `&is`, then attempts to extract and store an object of type `U`.

Types:

istream_iterator::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

istream_iterator::istream_type:

```
typedef basic_istream<E, T> istream_type;
```

The type is a synonym for `basic_istream<E, T>`.

istream_iterator::traits_type:

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

Operators:

istream_iterator::operator:*

```
const U& operator*() const;
```

The operator returns the stored object of type `U`.

istream_iterator::operator->:

```
const U *operator->() const;
```

The operator returns &**this.

istream_iterator::operator++:

```
istream_iterator<U, E, T, Dist>& operator++();
istream_iterator<U, E, T, Dist> operator++(int);
```

The first operator attempts to extract and store an object of type U from the associated input stream. The second operator makes a copy of the object, increments the object, then returns the copy.

istreambuf_iterator

Description: The template class describes an input iterator object. It extracts elements of class **E** from an **input stream buffer**, which it accesses via an object it stores, of type pointer to basic_streambuf<E, T>. After constructing or incrementing an object of class istreambuf_iterator with a non-null stored pointer, the object effectively attempts to extract and store an object of type E from the associated itput stream. (The extraction may be delayed, however, until the object is actually dereferenced or copied.) If the extraction fails, the object effectively replaces the stored pointer with a null pointer (thus making an end-of-sequence indicator).

Synopsis:

```
template<class E, class T = char_traits<E> >
    class istreambuf_iterator
        : public iterator<input_iterator_tag,
            E, typename T::off_type, E *, E&> {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef typename T::int_type int_type;
    typedef basic_streambuf<E, T> streambuf_type;
    typedef basic_istream<E, T> istream_type;
    istreambuf_iterator(streambuf_type *sb = 0) throw();
    istreambuf_iterator(istream_type& is) throw();
    const E& operator*() const;
    const E *operator->();
    istreambuf_iterator& operator++();
    istreambuf_iterator operator++(int);
    bool equal(const istreambuf_iterator& rhs) const;
    };
```

Constructor:

istreambuf_iterator::istreambuf_iterator:

```
istreambuf_iterator(streambuf_type *sb = 0) throw();
istreambuf_iterator(istream_type& is) throw();
```

The first constructor initializes the input stream-buffer pointer with sb. The second constructor initializes the input stream-buffer pointer with is.rdbuf(), then (eventually) attempts to extract and store an object of type E.

Types:

istreambuf_iterator::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

istreambuf_iterator::int_type:

```
typedef typename T::int_type int_type;
```

The type is a synonym for T::int_type.

istreambuf_iterator::istream_type:

```
typedef basic_istream<E, T> istream_type;
```

The type is a synonym for basic_istream<E, T>.

istreambuf_iterator::streambuf_type:

```
typedef basic_streambuf<E, T> streambuf_type;
```

The type is a synonym for basic_streambuf<E, T>.

istreambuf_iterator::traits_type:

```
typedef T traits_type;
```

The type is a synonym for the template parameter T.

Member functions:

istreambuf_iterator::equal:

```
bool equal(const istreambuf_iterator& rhs) const;
```

The member function returns true only if the stored stream buffer pointers for the object and rhs are both null pointers or are both non-null pointers.

Operators:

istreambuf_iterator::operator:*

```
const E& operator*() const;
```

The operator returns the stored object of type E.

istreambuf_iterator::operator++:

```
istreambuf_iterator& operator++();
istreambuf_iterator operator++(int);
```

The first operator (eventually) attempts to extract and store an object of type E from the associated input stream. The second operator makes a copy of the object, increments the object, then returns the copy.

istreambuf_iterator::operator->:

```
const E *operator->() const;
```

The operator returns &**this.

iterator

```
template<class C, class T, class Dist = ptrdiff_t
    class Pt = T *, class Rt = T&>
    struct iterator {
    typedef C iterator_category;
    typedef T value_type;
```

```
    typedef Dist difference_type;
    typedef Pt pointer;
    typedef Rt reference;
    };
```

The template class serves as a base type for all iterators. It defines the member types `iterator_category`, (a synonym for the template parameter `C`), `value_type` (a synonym for the template parameter `T`), `difference_type` (a synonym for the template parameter `Dist`), `pointer` (a synonym for the template parameter `Pt`), and `reference` (a synonym for the template parameter `T`).

Note that `value_type` should *not* be a constant type even if `pointer` points at an object of const type and `reference` designates an object of const type.

iterator_traits

```
template<class It>
    struct iterator_traits {
    typedef typename It::iterator_category iterator_category;
    typedef typename It::value_type value_type;
    typedef typename It::difference_type difference_type;
    typedef typename It::pointer pointer;
    typedef typename It::reference reference;
    };
template<class T>
    struct iterator_traits<T *> {
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T *pointer;
    typedef T& reference;
    };
template<class T>
    struct iterator_traits<const T *> {
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef const T *pointer;
    typedef const T& reference;
    };
```

The template class determines several critical types associated with the iterator type `It`. It defines the member types `iterator_category` (a synonym for `It::iterator_category`), `value_type` (a synonym for `It::value_type`), `difference_type` (a synonym for `It::difference_type`), `pointer` (a synonym for `It::pointer`), and `reference` (a synonym for `It::reference`).

The partial specializations determine the critical types associated with an object pointer type `T *`. In this implementation, you can also use several template functions that do not make use of partial specialization:

```
template<class C, class T, class Dist>
    C _Iter_cat(const iterator<C, T, Dist>&);
template<class T>
    random_access_iterator_tag _Iter_cat(const T *);

template<class C, class T, class Dist>
    T *_Val_type(const iterator<C, T, Dist>&);
template<class T>
    T *_Val_type(const T *);
```

```
template<class C, class T, class Dist>
    Dist *_Dist_type(const iterator<C, T, Dist>&);
template<class T>
    ptrdiff_t *_Dist_type(const T *);
```

which determine several of the same types a bit more indirectly. You use these functions as arguments on a function call. Their sole purpose is to supply a useful template class parameter to the called function.

ostream_iterator

Description: The template class describes an output iterator object. It inserts objects of class **U** into an **output stream**, which it accesses via an object it stores, of type pointer to `basic_ostream<E, T>`. It also stores a pointer to a **delimiter string**, a null-terminated string of elements of type `E`, which is appended after each insertion. (Note that the string itself is *not* copied by the constructor.

Synopsis:

```
template<class U, class E = char,
    class T = char_traits<E>  >
    class ostream_iterator
        : public iterator<output_iterator_tag,
            void, void, void, void> {
public:
    typedef U value_type;
    typedef E char_type;
    typedef T traits_type;
    typedef basic_ostream<E, T> ostream_type;
    ostream_iterator(ostream_type& os);
    ostream_iterator(ostream_type& os, const E *delim);
    ostream_iterator<U, E, T>& operator=(const U& val);
    ostream_iterator<U, E, T>& operator*();
    ostream_iterator<U, E, T>& operator++();
    ostream_iterator<U, E, T> operator++(int);
    };
```

Constructor:

ostream_iterator::ostream_iterator:

```
ostream_iterator(ostream_type& os);
ostream_iterator(ostream_type& os, const E *delim);
```

The first constructor initializes the output stream pointer with `&os`. The delimiter string pointer designates an empty string. The second constructor initializes the output stream pointer with `&os` and the delimiter string pointer with `delim`.

Types:

ostream_iterator::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

ostream_iterator::ostream_type:

```
typedef basic_ostream<E, T> ostream_type;
```

The type is a synonym for `basic_ostream<E, T>`.

ostream_iterator::traits_type:

```
typedef T traits_type;
```

The type is a synonym for the template parameter T.

ostream_iterator::value_type:

```
typedef U value_type;
```

The type is a synonym for the template parameter U.

Operators:

ostream_iterator::operator:*

```
ostream_iterator<U, E, T>& operator*();
```

The operator returns *this.

ostream_iterator::operator++:

```
ostream_iterator<U, E, T>& operator++();
ostream_iterator<U, E, T> operator++(int);
```

The operators both return *this.

ostream_iterator::operator=:

```
ostream_iterator<U, E, T>& operator=(const U& val);
```

The operator inserts val into the output stream associated with the object, then returns *this.

ostreambuf_iterator

Description: The template class describes an output iterator object. It inserts elements of class **E** into an **output stream buffer**, which it accesses via an object it stores, of type pointer to basic_streambuf<E, T>.

Synopsis:

```
template<class E, class T = char_traits<E> >
    class ostreambuf_iterator
        : public iterator<output_iterator_tag,
            void, void, void, void> {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef basic_streambuf<E, T> streambuf_type;
    typedef basic_ostream<E, T> ostream_type;
    ostreambuf_iterator(streambuf_type *sb) throw();
    ostreambuf_iterator(ostream_type& os) throw();
    ostreambuf_iterator& operator=(E x);
    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    T1 operator++(int);
    bool failed() const throw();
    };
```

Constructor:

ostreambuf_iterator::ostreambuf_iterator:

```
ostreambuf_iterator(streambuf_type *sb) throw();
ostreambuf_iterator(ostream_type& os) throw();
```

The first constructor initializes the output stream-buffer pointer with `sb`. The second constructor initializes the output stream-buffer pointer with `os.rdbuf()`. (The stored pointer must not be a null pointer.)

Types:

ostreambuf_iterator::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

ostreambuf_iterator::ostream_type:

```
typedef basic_ostream<E, T> ostream_type;
```

The type is a synonym for `basic_ostream<E, T>`.

ostreambuf_iterator::streambuf_type:

```
typedef basic_streambuf<E, T> streambuf_type;
```

The type is a synonym for `basic_streambuf<E, T>`.

ostreambuf_iterator::traits_type:

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

Member functions:

ostreambuf_iterator::failed:

```
bool failed() const throw();
```

The member function returns true only if no insertion into the output stream buffer has earlier failed.

Operators:

ostreambuf_iterator::operator:*

```
ostreambuf_iterator& operator*();
```

The operator returns `*this`.

ostreambuf_iterator::operator++:

```
ostreambuf_iterator& operator++();
T1 operator++(int);
```

The first operator returns `*this`. The second operator returns an object of some type `T1` that can be converted to `ostreambuf_iterator<E, T>`.

ostreambuf_iterator::operator=:

```
ostreambuf_iterator& operator=(E x);
```

The operator inserts `x` into the associated stream buffer, then returns `*this`.

reverse_iterator

Description: The template class describes an object that behaves like a random-access iterator, only in reverse. It stores a random-access iterator of type **RanIt** in the protected object **current**. Incrementing the object `x` of type `reverse_iterator` decrements `x.current`, and decrementing `x` increments `x.current`. Moreover, the expression `*x` evaluates to `*(current - 1)`, of type **Ref**. Typically, `Ref` is type `T&`.

Thus, you can use an object of class `reverse_iterator` to access in reverse order a sequence that is traversed in order by a random-access iterator.

Several STL containers specialize `reverse_iterator` for `RanIt` a bidirectional iterator. In these cases, you must not call any of the member functions `operator+=`, `operator+`, `operator-=`, `operator-`, or `operator[]`.

Synopsis:

```
template<class RanIt>
    class reverse_iterator : public iterator<
        typename iterator_traits<RanIt>::iterator_category,
        typename iterator_traits<RanIt>::value_type,
        typename iterator_traits<RanIt>::difference_type,
        typename iterator_traits<RanIt>::pointer,
        typename iterator_traits<RanIt>::reference> {
    typedef typename iterator_traits<RanIt>::difference_type
        Dist;
    typedef typename iterator_traits<RanIt>::pointer
        Ptr;
    typedef typename iterator_traits<RanIt>::reference
        Ref;
public:
    typedef RanIt iterator_type;
    reverse_iterator();
    explicit reverse_iterator(RanIt x);
    template<class U>
        reverse_iterator(const reverse_iterator<U>& x);
    RanIt base() const;
    Ref operator*() const;
    Ptr operator->() const;
    reverse_iterator& operator++();
    reverse_iterator operator++(int);
    reverse_iterator& operator--();
    reverse_iterator operator--();
    reverse_iterator& operator+=(Dist n);
    reverse_iterator operator+(Dist n) const;
    reverse_iterator& operator-=(Dist n);
    reverse_iterator operator-(Dist n) const;
    Ref operator[](Dist n) const;
protected:
    RanIt current;
    };
```

Constructor:

reverse_iterator::reverse_iterator:

```
reverse_iterator();
explicit reverse_iterator(RanIt x);
template<class U>
    reverse_iterator(const reverse_iterator<U>& x);
```

The first constructor initializes current with its default constructor. The second constructor initializes `current` with `x.current`.

The template constructor initializes `current` with `x.base ()`.

Types:

reverse_iterator::iterator_type:

```
typedef RanIt iterator_type;
```

The type is a synonym for the template parameter `RanIt`.

reverse_iterator::pointer:

```
typedef Ptr pointer;
```

The type is a synonym for the template parameter `Ref`.

reverse_iterator::reference:

```
typedef Ref reference;
```

The type is a synonym for the template parameter `Ref`.

Member functions:

reverse_iterator::base:

```
RanIt base() const;
```

The member function returns current.

Operators:

reverse_iterator::operator:*

```
Ref operator*() const;
```

The operator returns `*(current - 1)`.

reverse_iterator::operator+:

```
reverse_iterator operator+(Dist n) const;
```

The operator returns `reverse_iterator(*this) += n`.

reverse_iterator::operator++:

```
reverse_iterator& operator++();
reverse_iterator operator++(int);
```

The first (preincrement) operator evaluates `–current.` then returns `*this`.

The second (postincrement) operator makes a copy of `*this`, evaluates `–current`, then returns the copy.

reverse_iterator::operator+=:

```
reverse_iterator& operator+=(Dist n);
```

The operator evaluates `current - n`. then returns `*this`.

reverse_iterator::operator-:

```
reverse_iterator operator-(Dist n) const;
```

The operator returns `reverse_iterator(*this) -= n`.

reverse_iterator::operator—:

```
reverse_iterator& operator--();
reverse_iterator operator--();
```

The first (predecrement) operator evaluates `++current.` then returns `*this`.

The second (postdecrement) operator makes a copy of `*this`, evaluates `++current`, then returns the copy.

reverse_iterator::operator-=:

```
reverse_iterator& operator-=(Dist n);
```

The operator evaluates `current + n`. then returns `*this`.

reverse_iterator::operator->:

```
Ptr operator->() const;
```

The operator returns `&**this`.

reverse_iterator::operator[]:

```
Ref operator[](Dist n) const;
```

The operator returns `*(*this + n)`.

Template functions

advance

```
template<class InIt, class Dist>
    void advance(InIt& it, Dist n);
```

The template function effectively advances `it` by incrementing it `n` times. If `InIt` is a random-access iterator type, the function evaluates the expression `it += n`. Otherwise, it performs each increment by evaluating `++it`. If `InIt` is an input or forward iterator type, `n` must not be negative.

back_inserter

```
template<class Cont>
    back_insert_iterator<Cont> back_inserter(Cont& x);
```

The template function returns `back_insert_iterator<Cont>(x)`.

distance

```
template<class Init, class Dist>
    typename iterator_traits<InIt>::difference_type
        distance(InIt first, InIt last);
```

The template function sets a count `n` to zero. It then effectively advances `first` and increments `n` until `first == last`. If `InIt` is a random-access iterator type, the function evaluates the expression `n += last - first`. Otherwise, it performs each iterator increment by evaluating `++first`.

front_inserter

```
template<class Cont>
    front_insert_iterator<Cont> front_inserter(Cont& x);
```

The template function returns `front_insert_iterator<Cont>(x)`.

inserter

```
template<class Cont, class Iter>
    insert_iterator<Cont> inserter(Cont& x, Iter it);
```

The template function returns `insert_iterator<Cont>(x, it)`.

Types

bidirectional_iterator_tag

```
struct bidirectional_iterator_tag
    : public forward_iterator_tag {
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a bidirectional iterator.

forward_iterator_tag

```
struct forward_iterator_tag
    : public input_iterator_tag {
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a forward iterator.

input_iterator_tag

```
struct input_iterator_tag {
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as an input iterator.

output_iterator_tag

```
struct output_iterator_tag {
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a output iterator.

random_access_iterator_tag

```
struct random_access_iterator_tag
    : public bidirectional_iterator_tag {
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a random-access iterator.

Operators

operator!=

```
template<class RanIt>
    bool operator!=(
        const reverse_iterator<RanIt>& lhs,
```

```
        const reverse_iterator<RanIt>& rhs);
template<class U, class E, class T, class Dist>
    bool operator!=(
        const istream_iterator<U, E, T, Dist>& lhs,
        const istream_iterator<U, E, T, Dist>& rhs);
template<class E, class T>
    bool operator!=(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
```

The template operator returns `!(lhs == rhs)`.

operator==

```
template<class RanIt>
    bool operator==(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
template<class U, class E, class T, class Dist>
    bool operator==(
        const istream_iterator<U, E, T, Dist>& lhs,
        const istream_iterator<U, E, T, Dist>& rhs);
template<class E, class T>
    bool operator==(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
```

The first template operator returns true only if `lhs.current == rhs.current`. The second template operator returns true only if both `lhs` and `rhs` store the same stream pointer. The third template operator returns `lhs.equal(rhs)`.

operator<

```
template<class RanIt>
    bool operator<(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
```

The template operator returns `rhs.current < lhs.current` [sic].

operator<=

```
template<class RanIt>
    bool operator<=(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
```

The template operator returns `!(rhs < lhs)`.

operator>

```
template<class RanIt>
    bool operator>(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
```

The template operator returns `rhs < lhs`.

operator>=

```
template<class RanIt>
    bool operator>=(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
```

The template operator returns `!(lhs < rhs)`.

operator+

```
template<class RanIt>
    reverse_iterator<RanIt> operator+(Dist n,
        const reverse_iterator<RanIt>& rhs);
```

The template operator returns `rhs + n`.

operator-

```
template<class RanIt>
    Dist operator-(
        const reverse_iterator<RanIt>& lhs,
        const reverse_iterator<RanIt>& rhs);
```

The template operator returns `rhs.current - lhs.current` [sic].

<limits>

Description

Include the standard header **<limits>** to define the template class `numeric_limits`. Explicit specializations of this class describe many arithmetic properties of the scalar types (other than pointers).

Synopsis

```
namespace std {
enum float_denorm_style;
enum float_round_style;
template<class T>
    class numeric_limits;
    }
```

Enumerations

float_denorm_style

```
enum float_denorm_style {
    denorm_indeterminate = -1,
    denorm_absent = 0,
    denorm_present = 1
    };
```

The enumeration describes the various methods that an implementation can choose for representing a denormalized floating-point value — one too small to represent as a normalized value:

- **`denorm_indeterminate`** — presence or absence of denormalized forms cannot be determined at translation time
- **`denorm_absent`** — denormalized forms are absent
- **`denorm_present`** — denormalized forms are present

float_round_style

```
enum float_round_style {
    round_indeterminate = -1,
    round_toward_zero = 0,
    round_to_nearest = 1,
    round_toward_infinity = 2,
    round_toward_neg_infinity = 3
    };
```

The enumeration describes the various methods that an implementation can choose for rounding a floating-point value to an integer value:

- **round_indeterminate** — rounding method cannot be determined
- **round_toward_zero** — round toward zero
- **round_to_nearest** — round to nearest integer
- **round_toward_infinity** — round away from zero
- **round_toward_neg_infinity** — round to more negative integer

Classes

numeric_limits

Description: The template class describes many arithmetic properties of its parameter type T. The header defines explicit specializations for the types wchar_t, *bool, char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double,* and *long double.* For all these explicit specializations, the member is_specialized is true, and all relevant members have meaningful values. The program can supply additional explicit specializations.

For an arbitrary specialization, *no* members have meaningful values. A member object that does not have a meaningful value stores zero (or false) and a member function that does not return a meaningful value returns T(0).

Synopsis:

```
template<class T>
    class numeric_limits {
public:
    static const float_denorm_style has_denorm
        = denorm_absent;
    static const bool has_denorm_loss = false;
    static const bool has_infinity = false;
    static const bool has_quiet_NaN = false;
    static const bool has_signaling_NaN = false;
    static const bool is_bounded = false;
    static const bool is_exact = false;
    static const bool is_iec559 = false;
    static const bool is_integer = false;
    static const bool is_modulo = false;
    static const bool is_signed = false;
    static const bool is_specialized = false;
    static const bool tinyness_before = false;
    static const bool traps = false;
    static const float_round_style round_style =
        round_toward_zero;
    static const int digits = 0;
    static const int digits10 = 0;
    static const int max_exponent = 0;
    static const int max_exponent10 = 0;
    static const int min_exponent = 0;
    static const int min_exponent10 = 0;
    static const int radix = 0;
    static T denorm_min() throw();
    static T epsilon() throw();
    static T infinity() throw();
    static T max() throw();
    static T min() throw();
    static T quiet_NaN() throw();
    static T round_error() throw();
    static T signaling_NaN() throw();
    };
```

Members:

numeric_limits::digits:

```
static const int digits = 0;
```

The member stores the number of radix digits that the type can represent without change (which is the number of bits other than any sign bit for a predefined integer type, or the number of mantissa digits for a predefined floating-point type).

numeric_limits::digits10:

```
static const int digits10 = 0;
```

The member stores the number of decimal digits that the type can represent without change.

numeric_limits::has_denorm:

```
static const float_denorm_style has_denorm =
    denorm_absent;
```

The member stores denorm_present for a floating-point type that has denormalized values (effectively a variable number of exponent bits).

numeric_limits::has_denorm_loss:

```
static const bool has_denorm_loss = false;
```

The member stores true for a type that determines whether a value has lost accuracy because it is delivered as a denormalized result (too small to represent as a normalized value) or because it is inexact (not the same as a result not subject to limitations of exponent range and precision), an option with IEC 559 floating-point representations that can affect some results.

numeric_limits::has_infinity:

```
static const bool has_infinity = false;
```

The member stores true for a type that has a representation for positive infinity. True if is_iec559 is true.

numeric_limits::has_quiet_NaN:

```
static const bool has_quiet_NaN = false;
```

The member stores true for a type that has a representation for a **quiet NaN**, an encoding that is ``Not a Number'' which does not signal its presence in an expression. True if is_iec559 is true.

numeric_limits::has_signaling_NaN:

```
static const bool has_signaling_NaN = false;
```

The member stores true for a type that has a representation for a **signaling NaN**, an encoding that is ``Not a Number'' which signals its presence in an expression by reporting an exception. True if is_iec559 is true.

numeric_limits::is_bounded:

```
static const bool is_bounded = false;
```

The member stores true for a type that has a bounded set of representable values (which is the case for all predefined types).

numeric_limits::is_exact:

```
static const bool is_exact = false;
```

The member stores true for a type that has exact representations for all its values (which is the case for all predefined integer types). A fixed-point or rational representation is also considered exact, but not a floating-point representation.

numeric_limits::is_iec559:

```
static const bool is_iec559 = false;
```

The member stores true for a type that has a representation conforming to **IEC 559**, an international standard for representing floating-point values (also known as **IEEE 754** in the USA).

numeric_limits::is_integer:

```
static const bool is_integer = false;
```

The member stores true for a type that has an integer representation (which is the case for all predefined integer types).

numeric_limits::is_modulo:

```
static const bool is_modulo = false;
```

The member stores true for a type that has a **modulo representation**, where all results are reduced modulo some value (which is the case for all predefined unsigned integer types).

numeric_limits::is_signed:

```
static const bool is_signed = false;
```

The member stores true for a type that has a signed representation (which is the case for all predefined floating-point and signed integer types).

numeric_limits::is_specialized:

```
static const bool is_specialized = false;
```

The member stores true for a type that has an explicit specialization defined for template class numeric_limits (which is the case for all scalar types other than pointers).

numeric_limits::max_exponent:

```
static const int max_exponent = 0;
```

The member stores the maximum positive integer such that the type can represent as a finite value radix raised to that power (which is the value FLT_MAX_EXP for type *float*). Meaningful only for floating-point types.

numeric_limits::max_exponent10:

```
static const int max_exponent10 = 0;
```

The member stores the maximum positive integer such that the type can represent as a finite value 10 raised to that power (which is the value FLT_MAX_10_EXP for type *float*). Meaningful only for floating-point types.

numeric_limits::min_exponent:

```
static const int min_exponent = 0;
```

The member stores the minimum negative integer such that the type can represent as a normalized value radix raised to that power (which is the value FLT_MIN_EXP for type *float*). Meaningful only for floating-point types.

numeric_limits::min_exponent10:

```
static const int min_exponent10 = 0;
```

The member stores the minimum negative integer such that the type can represent as a normalized value 10 raised to that power (which is the value FLT_MIN_10_EXP for type *float*). Meaningful only for floating-point types.

numeric_limits::radix:

```
static const int radix = 0;
```

The member stores the base of the representation for the type (which is 2 for the predefined integer types, and the base to which the exponent is raised, or FLT_RADIX, for the predefined floating-point types).

numeric_limits::round_style:

```
static const float_round_style round_style =
     round_toward_zero;
```

The member stores a value that describes the vaious methods that an implementation can choose for rounding a floating-point value to an integer value.

numeric_limits::tinyness_before:

```
static const bool tinyness_before = false;
```

The member stores true for a type that determines whether a value is ``tiny'' (too small to represent as a normalized value) before rounding, an option with IEC 559 floating-point representations that can affect some results.

numeric_limits::traps:

```
static const bool traps = false;
```

The member stores true for a type that generates some kind of signal to report certain arithmetic exceptions.

Member functions:

numeric_limits::denorm_min:

```
static T denorm_min() throw();
```

The function returns the minimum value for the type (which is the same as min() if has_denorm is not equal to denorm_present).

numeric_limits::epsilon:

```
static T epsilon() throw();
```

The function returns the difference between 1 and the smallest value greater than 1 that is representable for the type (which is the value FLT_EPSILON for type *float*).

numeric_limits::infinity:

```
static T infinity() throw();
```

The function returns the representation of positive infinity for the type. The return value is meaningful only if has_infinity is true.

numeric_limits::max:

```
static T max() throw();
```

The function returns the maximum finite value for the type (which is INT_MAX for type *int* and FLT_MAX for type *float*). The return value is meaningful if is_bounded is true.

numeric_limits::min:

```
static T min() throw();
```

The function returns the minimum normalized value for the type (which is INT_MIN for type *int* and FLT_MIN for type *float*). The return value is meaningful if is_bounded is true or `is_bounded` is false and is_signed is false.

numeric_limits::quiet_NaN:

```
static T quiet_NaN() throw();
```

The function returns a representation of a quiet NaN for the type. The return value is meaningful only if has_quiet_NaN is true.

numeric_limits::round_error:

```
static T round_error() throw();
```

The function returns the maximum rounding error for the type.

numeric_limits::signaling_NaN:

```
static T signaling_NaN() throw();
```

The function returns a representation of a signaling NaN for the type. The return value is meaningful only if has_signaling_NaN is true.

<list>

Description

Include the STL standard header **<list>** to define the container template class `list` and several supporting templates.

Synopsis

```
namespace std {
template<class T, class A>
    class list;

        // TEMPLATE FUNCTIONS
template<class T, class A>
    bool operator==(
        const list<T, A>& lhs,
```

```
        const list<T, A>& rhs);
template<class T, class A>
    bool operator!=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator<(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator<=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    void swap(
        list<T, A>& lhs,
        list<T, A>& rhs);
    }
```

Classes

list

Description: The template class describes an object that controls a varying-length sequence of elements of type `T`. The sequence is stored as a bidirectional linked list of elements, each containing a member of type `T`.

The object allocates and frees storage for the sequence it controls through a stored allocator object of class `A`. Such an allocator object must have the same external interface as an object of template class allocator. Note that the stored allocator object is *not* copied when the container object is assigned.

List reallocation occurs when a member function must insert or erase elements of the controlled sequence. In all such cases, only iterators or references that point at erased portions of the controlled sequence become **invalid**.

All additions to the controlled sequence occur as if by calls to insert, which is the only member function that calls the constructor `T(const T&)`. If such an expression throws an exception, the container object inserts no new elements and rethrows the exception. Thus, an object of template class `list` is left in a known state when such exceptions occur.

Synopsis:

```
template<class T, class A = allocator<T> >
    class list {
public:
    typedef A allocator_type;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer
        const_pointer;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::value_type value_type;
    typedef T0 iterator;
```

```
typedef T1 const_iterator;
typedef T2 size_type;
typedef T3 difference_type;
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
typedef reverse_iterator<iterator>
    reverse_iterator;
list();
explicit list(const A& al);
explicit list(size_type n);
list(size_type n, const T& v);
list(size_type n, const T& v, const A& al);
list(const list& x);
template<class InIt>
    list(InIt first, InIt last);
template<class InIt>
    list(InIt first, InIt last, const A& al);
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
void resize(size_type n);
void resize(size_type n, T x);
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
reference front();
const_reference front() const;
reference back();
const_reference back() const;
void push_front(const T& x);
void pop_front();
void push_back(const T& x);
void pop_back();
template<class InIt>
    void assign(InIt first, InIt last);
void assign(size_type n, const T& x);
iterator insert(iterator it, const T& x);
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
void clear();
void swap(list& x);
void splice(iterator it, list& x);
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x, iterator first,
    iterator last);
void remove(const T& x);
templace<class Pred>
    void remove_if(Pred pr);
void unique();
template<class Pred>
    void unique(Pred pr);
void merge(list& x);
template<class Pred>
    void merge(list& x, Pred pr);
void sort();
```

```
    template<class Pred>
        void sort(Pred pr);
    void reverse();
    };
```

Constructor:

list::list:

```
list();
explicit list(const A& al);
explicit list(size_type n);
list(size_type n, const T& v);
list(size_type n, const T& v,
    const A& al);
list(const list& x);
template<class InIt>
    list(InIt first, InIt last);
template<class InIt>
    list(InIt first, InIt last, const A& al);
```

All constructors store an allocator object and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

The first two constructors specify an empty initial controlled sequence. The third constructor specifies a repetition of `n` elements of value `T()`. The fourth and fifth constructors specify a repetition of `n` elements of value `x`. The sixth constructor specifies a copy of the sequence controlled by `x`. If `InIt` is an integer type, the last two constructors specify a repetition of `(size_type)first` elements of value `(T)last`. Otherwise, the last two constructors specify the sequence `[first, last)`. None of the constructors perform any interim reallocations.

Types:

list::allocator_type:

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

list::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

list::const_pointer:

```
typedef typename A::const_pointer
    const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

list::const_reference:

```
typedef typename A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

list::const_reverse_iterator:

```
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

list::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

list::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T0.

list::pointer:

```
typedef typename A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

list::reference:

```
typedef typename A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

list::reverse_iterator:

```
typedef reverse_iterator<iterator>
    reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

list::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

list::value_type:

```
typedef typename A::value_type value_type;
```

The type is a synonym for the template parameter T.

Member functions:

list::assign:

```
template<class InIt>
    void assign(InIt first, InIt last);
void assign(size_type n, const T& x);
```

If InIt is an integer type, the first member function behaves the same as assign((size_type)first, (T)last). Otherwise, the first member function replaces the sequence controlled by *this with the sequence [first, last), which must *not* overlap the initial controlled sequence. The second member function replaces the sequence controlled by *this with a repetition of n elements of value x.

list::back:

```
reference back();
const_reference back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

list::begin:

```
const_iterator begin() const;
iterator begin();
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

list::clear:

```
void clear();
```

The member function calls erase(begin(), end()).

list::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

list::end:

```
const_iterator end() const;
iterator end();
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

list::erase:

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by it. The second member function removes the elements of the controlled sequence in the range [first, last). Both return an iterator that designates the first element remaining beyond any elements removed, or end() if no such element exists.

Erasing N elements causes N destructor calls. No reallocation occurs, so iterators and references become invalid only for the erased elements.

The member functions never throw an exception.

list::front:

```
reference front();
const_reference front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

list::get_allocator:

```
A get_allocator() const;
```

The member function returns the stored allocator object.

list::insert:

```
iterator insert(iterator it, const T& x);
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
```

Each of the member functions inserts, before the element pointed to by `it` in the controlled sequence, a sequence specified by the remaining operands. The first member function inserts a single element with value `x` and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of `n` elements of value `x`.

If `InIt` is an integer type, the last member function behaves the same as `insert(it, (size_type)first, (T)last)`. Otherwise, the last member function inserts the sequence `[first, last)`, which must *not* overlap the initial controlled sequence.

Inserting `N` elements causes `N` constructor calls. No reallocation occurs, so no iterators or references become invalid.

If an exception is thrown during the insertion of one or more elements, the container is left unaltered and the exception is rethrown.

list::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

list::merge:

```
void merge(list& x);
template<class Pred>
    void merge(list& x, Pred pr);
```

Both member functions remove all elements from the sequence controlled by `x` and insert them in the controlled sequence. Both sequences must be ordered by the same predicate, described below. The resulting sequence is also ordered by that predicate.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in *ascending* order.) The second member function imposes the order `!pr(*Pj, *Pi)` whenever `i < j`.

No pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence. If a pair of elements in the resulting controlled sequence compares equal (`!(*Pi < *Pj) && !(*Pj < *Pi)`), an element from the original controlled sequence appears before an element from the sequence controlled by `x`.

An exception occurs only if `pr` throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

list::pop_back:

```
void pop_back();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

The member function never throws an exception.

list::pop_front:

```
void pop_front();
```

The member function removes the first element of the controlled sequence, which must be non-empty.

The member function never throws an exception.

list::push_back:

```
void push_back(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

list::push_front:

```
void push_front(const T& x);
```

The member function inserts an element with value `x` at the beginning of the controlled sequence.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

list::rbegin:

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

list::remove:

```
void remove(const T& x);
```

The member function removes from the controlled sequence all elements, designated by the iterator `P`, for which `*P == x`.

The member function never throws an exception.

list::remove_if:

```
templace<class Pred>
    void remove_if(Pred pr);
```

The member function removes from the controlled sequence all elements, designated by the iterator P, for which pr(*P) is true.

An exception occurs only if pr throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

list::rend:

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

list::resize:

```
void resize(size_type n);
void resize(size_type n, T x);
```

The member functions both ensure that size() henceforth returns n. If it must make the controlled sequence longer, the first member function appends elements with value T(), while the second member function appends elements with value x. To make the controlled sequence shorter, both member functions call erase(begin() + n, end()).

list::reverse:

```
void reverse();
```

The member function reverses the order in which elements appear in the controlled sequence.

list::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

list::sort:

```
void sort();
template<class Pred>
    void sort(Pred pr);
```

Both member functions order the elements in the controlled sequence by a predicate, described below.

For the iterators Pi and Pj designating elements at positions i and j, the first member function imposes the order !(*Pj < *Pi) whenever i < j. (The elements are sorted in *ascending* order.) The member template function imposes the order !pr(*Pj, *Pi) whenever i < j. No pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence.

An exception occurs only if pr throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

list::splice:

```
void splice(iterator it, list& x);
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x, iterator first,
    iterator last);
```

The first member function inserts the sequence controlled by x before the element in the controlled sequence pointed to by it. It also removes all elements from x. (&x must not equal this.)

The second member function removes the element pointed to by first in the sequence controlled by x and inserts it before the element in the controlled sequence pointed to by it. (If it == first || it == ++first, no change occurs.)

The third member function inserts the subrange designated by [first, last) from the sequence controlled by x before the element in the controlled sequence pointed to by it. It also removes the original subrange from the sequence controlled by x. (If &x == this, the range [first, last) must not include the element pointed to by it.)

If the third member function inserts N elements, and &x != this, an object of class iterator is incremented N times. For all splice member functions, If get_allocator() == str.get_allocator(), no exception occurs. Otherwise, a copy and a destructor call also occur for each inserted element.

In all cases, only iterators or references that point at spliced elements become **invalid**.

list::swap:

```
void swap(list& x);
```

The member function swaps the controlled sequences between *this and x. If get_allocator() == x.get_allocator(), it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

list::unique:

```
void unique();
template<class Pred>
    void unique(Pred pr);
```

The first member function removes from the controlled sequence every element that compares equal to its preceding element. For the iterators Pi and Pj designating elements at positions i and j, the second member function removes every element for which i + 1 == j && pr(*Pi, *Pj).

For a controlled sequence of length N (> 0), the predicate pr(*Pi, *Pj) is evaluated N - 1 times.

An exception occurs only if pr throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

Template functions

operator!=

```
template<class T, class A>
    bool operator!=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class A>
    bool operator==(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function overloads `operator==` to compare two objects of template class list. The function returns `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end(), rhs.begin())`.

operator<

```
template<class T, class A>
    bool operator<(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `list`. The function returns `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class T, class A>
    bool operator<=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class A>
    bool operator>(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class A>
    bool operator>=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class T, class A>
    void swap(
        list <T, A>& lhs,
        list <T, A>& rhs);
```

The template function executes `lhs.swap (rhs)`.

<locale>

Description

Include the standard header **<locale>** to define a host of template classes and functions that encapsulate and manipulate locales.

Synopsis

```
namespace std {
class locale;
class ctype_base;
template<class E>
    class ctype;
template<>
    class ctype<char>;
template<class E>
    class ctype_byname;
class codect_base;
template<class From, class To, class State>
    class codecvt;
template<class From, class To, class State>
    class codecvt_byname;
template<class E, class InIt>
    class num_get;
template<class E, class OutIt>
    class num_put;
template<class E>
    class numpunct;
template<class E>
    class numpunct_byname;
template<class E>
    class collate;
template<class E>
    class collate_byname;
class time_base;
template<class E, class InIt>
    class time_get;
template<class E, class InIt>
    class time_get_byname;
template<class E, class OutIt>
    class time_put;
template<class E, class OutIt>
    class time_put_byname;
class money_base;
template<class E, bool Intl, class InIt>
    class money_get;
template<class E, bool Intl, class OutIt>
    class money_put;
template<class E, bool Intl>
    class moneypunct;
template<class E, bool Intl>
    class moneypunct_byname;
class messages_base;
template<class E>
    class messages;
template<class E>
    class messages_byname;

        // TEMPLATE FUNCTIONS
template<class Facet>
    bool has_facet(const locale& loc);
template<class Facet>
```

```
    const Facet& use_facet(const locale& loc);
template<class E>
    bool isspace(E c, const locale& loc) const;
template<class E>
    bool isprint(E c, const locale& loc) const;
template<class E>
    bool iscntrl(E c, const locale& loc) const;
template<class E>
    bool isupper(E c, const locale& loc) const;
template<class E>
    bool islower(E c, const locale& loc) const;
template<class E>
    bool isalpha(E c, const locale& loc) const;
template<class E>
    bool isdigit(E c, const locale& loc) const;
template<class E>
    bool ispunct(E c, const locale& loc) const;
template<class E>
    bool isxdigit(E c, const locale& loc) const;
template<class E>
    bool isalnum(E c, const locale& loc) const;
template<class E>
    bool isgraph(E c, const locale& loc) const;
template<class E>
    E toupper(E c, const locale& loc) const;
template<class E>
    E tolower(E c, const locale& loc) const;
    }
```

Classes

codecvt

Description: The template class describes an object that can serve as a locale facet, to control conversions between a sequence of values of type `From` and a sequence of values of type `To`. The class `State` characterizes the transformation — and an object of class `State` stores any necessary state information during a conversion.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

The template versions of `do_in` and `do_out` always return `codecvt_base::noconv`. The Standard C++ library defines an explicit specialization, however, that is more useful:

```
template<>
    codecvt<wchar_t, char, mbstate_t>
```

which converts between `wchar_t` and *char* sequences.

Synopsis:

```
template<class From, class To, class State>
    class codecvt
        : public locale::facet, codecvt_base {
public:
    typedef From intern_type;
    typedef To extern_type;
    typedef State state_type;
    explicit codecvt(size_t refs = 0);
    result in(State& state,
        const To *first1, const To *last1,
            const To *next1,
        From *first2, From *last2, From *next2);
```

```
    result out(State& state,
        const From *first1, const From *last1,
            const From *next1,
        To *first2, To *last2, To *next2);
    result unshift(State& state,
        To *first2, To *last2, To *next2);
    bool always_noconv() const throw();
    int max_length() const throw();
    int length(State& state,
        const To *first1, const To *last1,
            size_t _N2) const throw();
    int encoding() const throw();
    static locale::id id;
protected:
    ~codecvt();
    virtual result do_in(State& state,
        const To *first1, const To *last1,
            const To *next1,
        From *first2, From *last2, From *next2);
    virtual result do_out(State& state,
        const From *first1, const From *last1,
            const From *next1,
        To *first2, To *last2, To *next2);
    virtual result do_unshift(State& state,
        To *first2, To *last2, To *next2);
    virtual bool do_always_noconv() const throw();
    virtual int do_max_length() const throw();
    virtual int do_encoding() const throw();
    virtual int do_length(State& state,
        const To *first1, const To *last1,
            size_t len2) const throw();
    };
```

Constructor:

codecvt::codecvt:

`explicit codecvt(size_t refs = 0);`

The constructor initializes its `locale::facet` base object with `locale::facet(refs)`.

Types:

codecvt::extern_type:

`typedef To extern_type;`

The type is a synonym for the template parameter `To`.

codecvt::intern_type:

`typedef From intern_type;`

The type is a synonym for the template parameter `From`.

codecvt::state_type:

`typedef State state_type;`

The type is a synonym for the template parameter `State`.

Member functions:

codecvt::always_noconv:

`bool always_noconv() const throw();`

The member function returns do_always_noconv().

codecvt::in:

```
result in(State state&,
    const To *first1, const To *last1, const To *next1,
    From *first2, From *last2, From *next2);
```

The member function returns do_in(state, first1, last1, next1, first2, last2, next2).

codecvt::length:

```
int length(State state&,
    const To *first1, const To *last1,
        size_t len2) const throw();
```

The member function returns do_length(first1, last1, len2).

codecvt::encoding:

```
int encoding() const throw();
```

The member function returns do_encoding().

codecvt::max_length:

```
int max_length() const throw();
```

The member function returns do_max_length().

codecvt::out:

```
result out(State state&,
    const From *first1, const From *last1,
        const From *next1,
    To *first2, To *last2, To *next2);
```

The member function returns do_out(state, first1, last1, next1, first2, last2, next2).

codecvt::unshift:

```
result unshift(State state&,
    To *first2, To *last2, To *next2);
```

The member function returns do_unshift(state, first2, last2, next2).

codecvt::do_always_noconv:

```
virtual bool do_always_noconv() const throw();
```

The protected virtual member function returns true only if every call to do_in or do_out returns noconv. The template version always returns true.

codecvt::do_encoding:

```
virtual int do_encoding() const throw();
```

The protected virtual member function returns:
- -1, if the encoding of sequences of type extern_type is state dependent
- 0, if the encoding involves sequences of varying lengths
- n, if the encoding involves only sequences of length n

codecvt::do_in:

```
virtual result do_in(State state&,
    const To *first1, const To *last1, const To *next1,
    From *first2, From *last2, From *next2);
```

The protected virtual member function endeavors to convert the source sequence at `[first1, last1)` to a destination sequence that it stores within `[first2, last2)`. It always stores in `next1` a pointer to the first unconverted element in the source sequence, and it always stores in `next2` a pointer to the first unaltered element in the destination sequence.

`state` must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

The function returns:
- codecvt_base::error if the source sequence is ill formed
- codecvt_base::noconv if the function performs no conversion
- codecvt_base::ok if the conversion succeeds
- codecvt_base::partial if the source is insufficient, or if the destination is not large enough, for the conversion to succeed

The template version always returns `noconv`.

codecvt::do_length:

```
virtual int do_length(State state&,
    const To *first1, const To *last1,
        size_t len2) const throw();
```

The protected virtual member function effectively calls `do_in(state, first1, last1, next1, buf, buf + len2, next2)` for some buffer `buf` and pointers `next1` and `next2`, then returns `next2 - buf`. (Thus, it counts the maximum number of conversions, not greater than `len2`, defined by the source sequence at `[first1, last1)`.)

The template version always returns the lesser of `last1 - first1` and `len2`.

codecvt::do_max_length:

```
virtual int do_max_length() const throw();
```

The protected virtual member function returns the largest permissible value that can be returned by `do_length(first1, last1, 1)`, for arbitrary valid values of `first1` and `last1`. (Thus, it is roughly analogous to the macro MB_CUR_MAX, at least when `To` is type `char`.)

The template version always returns 1.

codecvt::do_out:

```
virtual result do_out(State state&,
    const From *first1, const From *last1,
        const From *next1,
    To *first2, To *last2, To *next2);
```

The protected virtual member function endeavors to convert the source sequence at `[first1, last1)` to a destination sequence that it stores within `[first2, last2)`. It always stores in next1 a pointer to the first unconverted element in the source sequence, and it always stores in next2 a pointer to the first unaltered element in the destination sequence.

state must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

The function returns:
- codecvt_base::error if the source sequence is ill formed
- codecvt_base::noconv if the function performs no conversion
- codecvt_base::ok if the conversion succeeds
- codecvt_base::partial if the source is insufficient, or if the destination is not large enough, for the conversion to succeed

The template version always returns noconv.

codecvt::do_unshift:

```
virtual result do_unshift(State state&,
    To *first2, To *last2, To *next2);
```

The protected virtual member function endeavors to convert the source element From(0) to a destination sequence that it stores within [first2, last2), except for the terminating element To(0). It always stores in next2 a pointer to the first unaltered element in the destination sequence.

state must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. Typically, converting the source element From(0) leaves the current state in the initial conversion state.

The function returns:
- codecvt_base::error if state represents an invalid state
- codecvt_base::noconv if the function performs no conversion
- codecvt_base::ok if the conversion succeeds
- codecvt_base::partial if the destination is not large enough for the conversion to succeed

The template version always returns noconv.

codecvt_base

Description: The class describes an enumeration common to all specializations of template class codecvt. The enumeration **result** describes the possible return values from do_in or do_out:
- **error** if the source sequence is ill formed
- **noconv** if the function performs no conversion
- **ok** if the conversion succeeds
- **partial** if the destination is not large enough for the conversion to succeed

Synopsis:

```
class codecvt_base {
public:
    enum result {ok, partial, error, noconv};
    };
```

codecvt_byname

Description: The template class describes an object that can serve as a locale facet of type `codecvt<From, To, State>`. Its behavior is determined by the named locale `s`. The constructor initializes its base object with `codecvt<From, To, State>(refs)`.

Synopsis:

```
template<class From, class To, class State>
    class codecvt_byname
        : public codecvt<From, To, State> {
public:
    explicit codecvt_byname(const char *s,
        size_t refs = 0);
protected:
    ~codecvt_byname();
    };
```

collate

Description: The template class describes an object that can serve as a locale facet, to control comparisons of sequences of type `E`.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

Synopsis:

```
template<class E>
    class collate : public locale::facet {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
    explicit collate(size_t refs = 0);
    int compare(const E *first1, const E *last1,
        const E *first2, const E *last2) const;
    string_type transform(const E *first,
        const E *last) const;
    long hash(const E *first, const E *last) const;
    static locale::id id;
protected:
    ~collate();
    virtual int
        do_compare(const E *first1, const E *last1,
            const E *first2, const E *last2) const;
    virtual string_type do_transform(const E *first,
        const E *last) const;
    virtual long do_hash(const E *first,
        const E *last) const;
    };
```

Constructor:

collate::collate:

```
explicit collate(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

Types:

collate::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

collate::string_type:

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class basic_string whose objects can store copies of the source sequence.

Member functions:

collate::compare:

```
int compare(const E *first1, const E *last1,
    const E *first2, const E *last2) const;
```

The member function returns `do_compare(first1, last1, first2, last2)`.

collate::hash:

```
long hash(const E *first, const E *last) const;
```

The member function returns `do_hash(first, last)`.

collate::transform:

```
string_type transform(const E *first,
    const E *last) const;
```

The member function returns `do_transform(first, last)`.

collate::do_compare:

```
virtual int do_compare(const E *first1, const E *last1,
    const E *first2, const E *last2) const;
```

The protected virtual member function compares the sequence at `[first1, last1)` with the sequence at `[first2, last2)`. It compares values by applying `operator<` between pairs of corresponding elements of type E. The first sequence compares less if it has the smaller element in the earliest unequal pair in the sequences, or if no unequal pairs exist but the first sequence is shorter.

If the first sequence compares less than the second sequence, the function returns -1. If the second sequence compares less, the function returns +1. Otherwise, the function returns zero.

collate::do_hash:

```
virtual long do_hash(const E *first,
    const E *last) const;
```

The protected virtual member function returns an integer derived from the values of the elements in the sequence `[first, last)`. Such a **hash** value can be useful, for example, in distributing sequences pseudo randomly across an array of lists.

collate::do_transform:

```
virtual string_type do_transform(const E *first,
    const E *last) const;
```

The protected virtual member function returns an object of class string_type whose controlled sequence is a copy of the sequence `[first, last)`. If a class derived from `collate<E>` overrides do_compare, it should also override `do_transform` to match. Put simply, two transformed strings should yield the same result, when passed to `collate::compare`, that you would get from passing the untransformed strings to `compare` in the derived class.

collate_byname

Description: The template class describes an object that can serve as a locale facet of type `collate<E>`. Its behavior is determined by the named locale `s`. The constructor initializes its base object with `collate<E>(refs)`.

Synopsis:

```
template<class E>
    class collate_byname : public collate<E> {
public:
    explicit collate_byname(const char *s,
        size_t refs = 0);
protected:
    ~collate_byname();
    };
```

ctype

Description: The template class describes an object that can serve as a locale facet, to characterize various properties of a ``character'' (element) of type `E`. Such a facet also converts between sequences of `E` elements and sequences of *char.*

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

The Standard C++ library defines two explicit specializations of this template class:
- ctype<char>, an explicit specialization whose differences are described separately
- `ctype<wchar_t>`, which treats elements as wide characters

In this implementation, other specializations of template class `ctype<E>`:
- convert a value `ch` of type `E` to a value of type *char* with the expression `(char)ch`
- convert a value `c` of type *char* to a value of type `E` with the expression `E(c)`

All other operations are performed on *char* values the same as for the explicit specialization `ctype<char>`.

Synopsis:

```
template<class E>
    class ctype
        : public locale::facet, public ctype_base {
public:
    typedef E char_type;
    explicit ctype(size_t refs = 0);
    bool is(mask msk, E ch) const;
    const E *is(const E *first, const E *last,
        mask *dst) const;
    const E *scan_is(mask msk, const E *first,
        const E *last) const;
    const E *scan_not(mask msk, const E *first,
        const E *last) const;
    E toupper(E ch) const;
    const E *toupper(E *first, E *last) const;
```

```
    E tolower(E ch) const;
    const E *tolower(E *first, E *last) const;
    E widen(char ch) const;
    const char *widen(char *first, char *last,
        E *dst) const;
    char narrow(E ch, char dflt) const;
    const E *narrow(const E *first, const E *last,
        char dflt, char *dst) const;
    static locale::id id;
protected:
    ~ctype();
    virtual bool do_is(mask msk, E ch) const;
    virtual const E *do_is(const E *first, const E *last,
        mask *dst) const;
    virtual const E *do_scan_is(mask msk, const E *first,
        const E *last) const;
    virtual const E *do_scan_not(mask msk, const E *first,
        const E *last) const;
    virtual E do_toupper(E ch) const;
    virtual const E *do_toupper(E *first, E *last) const;
    virtual E do_tolower(E ch) const;
    virtual const E *do_tolower(E *first, E *last) const;
    virtual E do_widen(char ch) const;
    virtual const char *do_widen(char *first, char *last,
        E *dst) const;
    virtual char do_narrow(E ch, char dflt) const;
    virtual const E *do_narrow(const E *first,
        const E *last, char dflt, char *dst) const;
    };
```

Constructor:

ctype::ctype:

```
explicit ctype(size_t refs = 0);
```

The constructor initializes its `locale::facet` base object with `locale::facet(refs)`.

Types:

ctype::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

Member functions:

ctype::do_is:

```
virtual bool do_is(mask msk, E ch) const;
virtual const E *do_is(const E *first, const E *last,
    mask *dst) const;
```

The first protected member template function returns true if `MASK(ch) & msk` is nonzero, where `MASK(ch)` designates the mapping between an element value `ch` and its classification mask, of type mask. The name `MASK` is purely symbolic here; it is not defined by the template class. For an object of class ctype<char>, the mapping is `tab[(unsigned char)(char)ch]`, where `tab` is the stored pointer to the ctype mask table.

The second protected member template function stores in `dst[I]` the value `MASK(first[I]) & msk`, where `I` ranges over the interval `[0, last - first)`.

ctype::do_narrow:

```
virtual char do_narrow(E ch, char dflt) const;
virtual const E *do_narrow(const E *first, const E *last,
    char dflt, char *dst) const;
```

The first protected member template function returns `(char)ch`, or `dflt` if that expression is undefined.

The second protected member template function stores in `dst[I]` the value `do_narrow(first[I], dflt)`, for `I` in the interval `[0, last - first)`.

ctype::do_scan_is:

```
virtual const E *do_scan_is(mask msk, const E *first,
    const E *last) const;
```

The protected member function returns the smallest pointer `p` in the range `[first, last)` for which `do_is(msk, *p)` is true. If no such value exists, the function returns `last`.

ctype::do_scan_not:

```
virtual const E *do_scan_not(mask msk, const E *first,
    const E *last) const;
```

The protected member function returns the smallest pointer `p` in the range `[first, last)` for which `do_is(msk, *p)` is false. If no such value exists, the function returns `last`.

ctype::do_tolower:

```
virtual E do_tolower(E ch) const;
virtual const E *do_tolower(E *first, E *last) const;
```

The first protected member template function returns the lowercase character corresponding to `ch`, if such a character exists. Otherwise, it returns `ch`.

The second protected member template function replaces each element `first[I]`, for `I` in the interval `[0, last - first)`, with `do_tolower(first[I])`.

ctype::do_toupper:

```
virtual E do_toupper(E ch) const;
virtual const E *do_toupper(E *first, E *last) const;
```

The first protected member template function returns the uppercase character corresponding to `ch`, if such a character exists. Otherwise, it returns `ch`.

The second protected member template function replaces each element `first[I]`, for `I` in the interval `[0, last - first)`, with `do_toupper(first[I])`.

ctype::do_widen:

```
virtual E do_widen(char ch) const;
virtual const char *do_widen(char *first, char *last,
    E *dst) const;
```

The first protected member template function returns `E(ch)`.

The second protected member template function stores in `dst[I]` the value `do_widen(first[I])`, for `I` in the interval `[0, last - first)`.

ctype::is:

```
bool is(mask msk, E ch) const;
const E *is(const E *first, const E *last,
    mask *dst) const;
```

The first member function returns `do_is(msk, ch)`. The second member function returns `do_is(first, last, dst)`.

ctype::narrow:

```
char narrow(E ch, char dflt) const;
const E *narrow(const E *first, const E *last,
    char dflt, char *dst) const;
```

The first member function returns `do_narrow(ch, dflt)`. The second member function returns `do_narrow(first, last, dflt, dst)`.

ctype::scan_is:

```
const E *scan_is(mask msk, const E *first,
    const E *last) const;
```

The member function returns `do_scan_is(msk, first, last)`.

ctype::scan_not:

```
const E *scan_not(mask msk, const E *first,
    const E *last) const;
```

The member function returns `do_scan_not(msk, first, last)`.

ctype::tolower:

```
E tolower(E ch) const;
const E *tolower(E *first, E *last) const;
```

The first member function returns `do_tolower(ch)`. The second member function returns `do_tolower(first, last)`.

ctype::toupper:

```
E toupper(E ch) const;
const E *toupper(E *first, E *last) const;
```

The first member function returns `do_toupper(ch)`. The second member function returns `do_toupper(first, last)`.

ctype::widen:

```
E widen(char ch) const;
const char *widen(char *first, char *last, E *dst) const;
```

The first member function returns `do_widen(ch)`. The second member function returns `do_widen(first, last, dst)`.

ctype<char>

Description: The class is an explicit specialization of template class ctype for type *char*. Hence, it describes an object that can serve as a locale facet, to characterize various properties of a ``character'' (element) of type *char*. The explicit specialization differs from the template class in several ways:

- An object of class ctype<char> stores a pointer to the first element of a **ctype mask table**, an array of UCHAR_MAX + 1 elements of type ctype_base::mask. It also stores a boolean object that indicates whether the array should be deleted when the ctype<E> object is destroyed.
- Its sole public constructor lets you specify tab, the ctype mask table, and del, the boolean object that is true if the array should be deleted when the ctype<char> object is destroyed — as well as the usual reference-count parameter refs.
- The protected member function table() returns the stored ctype mask table.
- The static member object table_size specifies the minimum number of elements in a ctype mask table.
- The protected static member function classic_table() returns the ctype mask table appropriate to the "C" locale.
- There are no protected virtual member functions do_is, do_scan_is, or do_scan_not. The corresponding public member functions perform the equivalent operations themselves.
- The member functions do_narrow and do_widen simply copy elements unaltered.

Synopsis:

```
template<>
    class ctype<char>
    : public locale::facet, public ctype_base {
public:
    typedef char char_type;
    explicit ctype(const mask *tab = 0, bool del = false,
        size_t refs = 0);
    bool is(mask msk, char ch) const;
    const char *is(const char *first, const char *last,
        mask *dst) const;
    const char *scan_is(mask msk,
        const char *first, const char *last) const;
    const char *scan_not(mask msk,
        const char *first, const char *last) const;
    char toupper(char ch) const;
    const char *toupper(char *first, char *last) const;
    char tolower(char ch) const;
    const char *tolower(char *first, char *last) const;
    char widen(char ch) const;
    const char *widen(char *first, char *last,
        char *dst) const;
    char narrow(char ch, char dflt) const;
    const char *narrow(const char *first,
        const char *last, char dflt, char *dst) const;
    static locale::id id;
protected:
    ~ctype();
    virtual char do_toupper(char ch) const;
    virtual const char *do_toupper(char *first,
        char *last) const;
    virtual char do_tolower(char ch) const;
    virtual const char *do_tolower(char *first,
        char *last) const;
    virtual char do_widen(char ch) const;
    virtual const char *do_widen(char *first, char *last,
        char *dst) const;
    virtual char do_narrow(char ch, char dflt) const;
    virtual const char *do_narrow(const char *first,
        const char *last, char dflt, char *dst) const;
```

```
    const mask *table() const throw();
    static const mask *classic_table() const throw();
    static const size_t table_size;
    };
```

ctype_base

Description: The class serves as a base class for facets of template class ctype. It defines just the enumerated type **mask** and several constants of this type. Each of the constants characterizes a different way to classify characters, as defined by the functions with similar names declared in the header <ctype.h>. The constants are:

- **space** (function isspace)
- **print** (function isprint)
- **cntrl** (function iscntrl)
- **upper** (function isupper)
- **lower** (function islower)
- **digit** (function isdigit)
- **punct** (function ispunct)
- **xdigit** (function isxdigit)
- **alpha** (function isalpha)
- **alnum** (function isalnum)
- **graph** (function isgraph)

You can charaterize a combination of classifications by ORing these constants. In particular, it is always true that `alnum == (alpha | digit)` and `graph == (alnum | punct)`.

Synopsis:

```
class ctype_base {
public:
    enum mask;
    static const mask space, print, cntrl,
        upper, lower, digit, punct, xdigit,
        alpha, alnum, graph;
    };
```

ctype_byname

Description: The template class describes an object that can serve as a locale facet of type `ctype<E>`. Its behavior is determined by the named locale `s`. The constructor initializes its base object with `ctype<E>(refs)` (or the equivalent for base class ctype<char>).

Synopsis:

```
template<class E>
    class ctype_byname : public ctype<E> {
public:
    explicit ctype_byname(const char *s,
        size_t refs = 0);
protected:
    ~ctype_byname();
    };
```

locale

Description: The class describes a **locale object** that encapsulates a locale. It represents culture-specific information as a list of **facets**. A facet is a pointer to an object of a class derived from class facet that has a public object of the form:

```
static locale::id id;
```

You can define an open-ended set of these facets. You can also construct a locale object that designates an arbitrary number of facets.

Predefined groups of these facets represent the locale categories traditionally managed in the Standard C library by the function setlocale.

Category **collate** (LC_COLLATE) includes the facets:

```
collate<char>
collate<wchar_t>
```

Category **ctype** (LC_CTYPE) includes the facets:

```
ctype<char>
ctype<wchar_t>
codecvt<char, char, mbstate_t>
codecvt<wchar_t, char, mbstate_t>
```

Category **monetary** (LC_MONETARY) includes the facets:

```
moneypunct<char, false>
moneypunct<wchar_t, false>
moneypunct<char, true>
moneypunct<wchar_t, true>
money_get<char, istreambuf_iterator<char> >
money_get<wchar_t, istreambuf_iterator<wchar_t> >
money_put<char, ostreambuf_iterator<char> >
money_put<wchar_t, ostreambuf_iterator<wchar_t> >
```

Category **numeric** (LC_NUMERIC) includes the facets:

```
num_get<char, istreambuf_iterator<char> >
num_get<wchar_t, istreambuf_iterator<wchar_t> >
num_put<char, ostreambuf_iterator<char> >
num_put<wchar_t, ostreambuf_iterator<wchar_t> >
numpunct<char>
numpunct<wchar_t>
```

Category **time** (LC_TIME) includes the facets:

```
time_get<char, istreambuf_iterator<char> >
time_get<wchar_t, istreambuf_iterator<wchar_t> >
time_put<char, ostreambuf_iterator<char> >
time_put<wchar_t, ostreambuf_iterator<wchar_t> >
```

Category **messages** [sic] (`LC_MESSAGE`) includes the facets:

```
messages<char>
messages<wchar_t>
```

(The last category is required by Posix, but not the C Standard.)

Some of these predefined facets are used by the iostreams classes, to control the conversion of numeric values to and from text sequences.

An object of class `locale` also stores a **locale name** as an object of class string. Using an invalid locale name to construct a locale facet or a locale object throws an object of class runtime_error. The stored locale name is "*" if the locale object cannot be certain that a C-style locale corresponds exactly to that represented by the object. Otherwise, you can establish a matching locale within the Standard C library, for the locale object x, by calling setlocale(LC_ALL, x.name. c_str()).

In this implementation, you can also call the static member function:

```
static locale empty();
```

to construct a locale object that has no facets. It is also a **transparent locale —** if the template functions has_facet and use_facet cannot find the requested facet in a transparent locale, they consult first the global locale and then, if that is transparent, the classic locale. Thus, you can write:

```
cout.imbue(locale::empty());
```

Subsequent insertions to cout are mediated by the current state of the global locale. You can even write:

```
locale loc(locale::empty(), locale::classic(),
    locale::numeric);
cout.imbue(loc);
```

Numeric formatting rules for subsequent insertions to cout remain the same as in the C locale, even as the global locale supplies changing rules for inserting dates and monetary amounts.

Synopsis:

```
class locale {
public:
    class facet;
    class id;
    typedef int category;
    static const category none, collate, ctype, monetary,
        numeric, time, messages, all;
    locale();
    explicit locale(const char *s);
    locale(const locale& x, const locale& y,
        category cat);
    locale(const locale& x, const char *s, category cat);
    template<class Facet>
        locale(const locale& x, Facet *fac);
    template<class Facet>
        locale combine(const locale& x) const;
    template<class E, class T, class A>
        bool operator()(const basic_string<E, T, A>& lhs,
            const basic_string<E, T, A>& rhs) const;
    string name() const;
    bool operator==(const locale& x) const;
    bool operator!=(const locale& x) const;
    static locale global(const locale& x);
    static const locale& classic();
    };
```

Constructor:

locale::locale:

```
locale();
explicit locale(const char *s);
locale(const locale& x, const locale& y,
    category cat);
locale(const locale& x, const char *s, category cat);
template<class Facet>
    locale(const locale& x, Facet *fac);
```

The first constructor initializes the object to match the global locale. The second constructor initializes all the locale categories to have behavior consistent with the locale name `s`. The remaining constructors copy `x`, with the exceptions noted:

```
locale(const locale& x, const locale& y,
    category cat);
```

replaces from `y` those facets corresponding to a category `c` for which `c & cat` is nonzero.

```
locale(const locale& x, const char *s, category cat);
```

replaces from `locale(s, all)` those facets corresponding to a category `c` for which `c & cat` is nonzero.

```
template<class Facet>
    locale(const locale& x, Facet *fac);
```

replaces in (or adds to) `x` the facet `fac`, if `fac` is not a null pointer.

If a locale name `s` is a null pointer or otherwise invalid, the function throws runtime_error.

Types:

locale::category:

```
typedef int category;
static const category none, collate, ctype, monetary,
    numeric, time, messages, all;
```

The type is a synonym for *int*, so that it can represent any of the C locale categories. It can also represent a group of constants local to class `locale`:

- **none**, corresponding to none of the the C categories
- **collate**, corresponding to the C category LC_COLLATE
- **ctype**, corresponding to the C category LC_CTYPE
- **monetary**, corresponding to the C category LC_MONETARY
- **numeric**, corresponding to the C category LC_NUMERIC
- **time**, corresponding to the C category LC_TIME
- **messages**, corresponding to the Posix category `LC_MESSAGE`
- **all**, corresponding to the C union of all categories LC_ALL

You can represent an arbitrary group of categories by ORing these constants, as in `monetary | time`.

Member classes:

locale::facet:

```
class facet {
protected:
    explicit facet(size_t refs = 0);
    virtual ~facet();
private:
    facet(const facet&)          // not defined
    void operator=(const facet&) // not defined
    };
```

The member class serves as the base class for all locale facets. Note that you can neither copy nor assign an object of class `facet`. You can construct and destroy objects derived from class `locale::facet`, but not objects of the base class proper. Typically, you construct an object `myfac` derived from `facet` when you construct a locale, as in:

```
locale loc(locale::classic(), new myfac);
```

In such cases, the constructor for the base class `facet` should have a zero `refs` argument. When the object is no longer needed, it is deleted. Thus, you supply a nonzero `refs` argument only in those rare cases where you take responsibility for the lifetime of the object.

locale::id:

```
class id {
protected:
    id();
private:
    id(const id&)             // not defined
    void operator=(const id&) // not defined
    };
```

The member class describes the static member object required by each unique locale facet. Note that you can neither copy nor assign an object of class `id`.

Member functions:

locale::classic:

```
static const locale& classic();
```

The static member function returns a locale object that represents the **classic locale**, which behaves the same as the C locale within the Standard C library.

locale::combine:

```
template<class Facet>
    locale combine(const locale& x) const;
```

The member function returns a locale object that replaces in (or adds to) `*this` the facet `Facet` listed in `x`.

locale::global:

```
static locale global(const locale& x);
```

The static member function stores a copy of `x` as the **global locale**. It also calls `setlocale( LC_ALL, x.name. c_str())`, to establishing a matching locale within the Standard C library. The function then returns the previous global locale. At program startup, the global locale is the same as the classic locale.

locale::name:

```
string name() const;
```

The member function returns the stored locale name.

locale::operator!=:

```
bool operator!=(const locale& x) const;
```

The member function returns `!(*this == x)`.

locale::operator():

```
template<class E, class T, class A>
    bool operator()(const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
```

The member function effectively executes:

```
const collate<E>& fac = use_fac<collate<E> >(*this);
return (fac.compare(lhs.begin(), lhs.end(),
    rhs.begin(), rhs.end()) < 0);
```

Thus, you can use a locale object as a function object.

locale::operator==:

```
bool operator==(const locale& x) const;
```

The member function returns true only if `*this` and `x` are copies of the same locale or have the same name (other than "*").

messages

Description: The template class describes an object that can serve as a locale facet, to characterize various properties of a **message catalog** that can supply messages represented as sequences of elements of type `E`.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

Synopsis:

```
template<class E>
    class messages
        : public locale::facet, public messages_base {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
    explicit messages(size_t refs = 0);
    catalog open(const string& name,
        const locale& loc) const;
    string_type get(catalog cat, int set, int msg,
        const string_type& dflt) const;
    void close(catalog cat) const;
    static locale::id id;
protected:
    ~messages();
    virtual catalog do_open(const string& name,
        const locale& loc) const;
    virtual string_type do_get(catalog cat, int set,
        int msg, const string_type& dflt) const;
    virtual void do_close(catalog cat) const;
    };
```

Constructor:

messages::messages:

```
explicit messages(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

Types:

messages::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

messages::string_type:

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class basic_string whose objects can store copies of the message sequences.

Member functions:

messages::close:

```
void close(catalog cat) const;
```

The member function calls do_close(cat);.

messages::do_close:

```
virtual void do_close(catalog cat) const;
```

The protected member function closes the message catalog cat, which must have been opened by an earlier call to do_open.

messages::do_get:

```
virtual string_type do_get(catalog cat, int set, int msg,
    const string_type& dflt) const;
```

The protected member function endeavors to obtain a message sequence from the message catalog cat. It may make use of set, msg, and dflt in doing so. It returns a copy of dflt on failure. Otherwise, it returns a copy of the specified message sequence.

messages::do_open:

```
virtual catalog do_open(const string& name,
    const locale& loc) const;
```

The protected member function endeavors to open a message catalog whose name is name. It may make use of the locale loc in doing so. It returns a value that compares less than zero on failure. Otherwise, the returned value can be used as the first argument on a later call to get. It should in any case be used as the argument on a later call to close.

messages::get:

```
string_type get(catalog cat, int set, int msg,
    const string_type& dflt) const;
```

The member function returns do_get(cat, set, msg, dflt);.

messages::open:

```
catalog open(const string& name,
    const locale& loc) const;
```

The member function returns do_open(name, loc);.

messages_base

Description: The class describes a type common to all specializations of template class messages. The type **catalog** is a synonym for type *int* that describes the possible return values from `messages::do_open`.

Synopsis:

```
class messages_base {
    typedef int catalog;
    };
```

messages_byname

Description: The template class describes an object that can serve as a `locale facet` of type `messages<E>`. Its behavior is determined by the named locale `s`. The constructor initializes its base object with `messages<E>(refs)`.

Synopsis:

```
template<class E>
    class messages_byname : public messages<E> {
public:
    explicit messages_byname(const char *s,
        size_t refs = 0);
protected:
    ~messages_byname();
    };
```

money_base

Description: The class describes an enumeration and a structure common to all specializations of template class moneypunct. The enumeration **part** describes the possible values in elements of the array **field** in the structure **pattern**. The values of `part` are:

- **none** to match zero or more spaces or generate nothing
- **sign** to match or generate a positive or negative sign
- **space** to match zero or more spaces or generate a space
- **symbol** to match or generate a currency symbol
- **value** to match or generate a monetary value

Synopsis:

```
class money_base {
    enum part {none, sign, space, symbol, value};
    struct pattern {
        char field[4];
        };
    };
```

money_get

Description: The template class describes an object that can serve as a locale facet, to control conversions of sequences of type `E` to monetary values.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

Synopsis:

```
template<class E,
    class InIt = istreambuf_iterator<E> >
    class money_get : public locale::facet {
public:
    typedef E char_type;
    typedef InIt iter_type;
    typedef basic_string<E> string_type;
    explicit money_get(size_t refs = 0);
    iter_type get(iter_type first, iter_type last,
        bool intl, ios_base& x, ios_base::iostate& st,
            long double& val) const;
    iter_type get(iter_type first, iter_type last,
        bool intl, ios_base& x, ios_base::iostate& st,
            string_type& val) const;
    static locale::id id;
protected:
    ~money_get();
    virtual iter_type do_get(iter_type first,
        iter_type last, bool intl, ios_base& x,
        ios_base::iostate& st, string_type& val) const;
    virtual iter_type do_get(iter_type first,
        iter_type last, bool intl, ios_base& x,
        ios_base::iostate& st, long double& val) const;
    };
```

Constructor:

money_get::money_get:

```
explicit money_get(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

Types:

money_get::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

money_get::iter_type:

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter `InIt`.

money_get::string_type:

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class basic_string whose objects can store sequences of elements from the source sequence.

Member functions:

money_get::do_get:

```
virtual iter_type do_get(iter_type first, iter_type last,
    bool intl, ios_base& x, ios_base::iostate& st,
        string_type& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
    bool intl, ios_base& x, ios_base::iostate& st,
        long double& val) const;
```

The first virtual protected member function endeavors to match sequential elements beginning at first in the sequence [first, last) until it has recognized a complete, nonempty **monetary input field**. If successful, it converts this field to a sequence of one or more decimal digits, optionally preceded by a minus sign (-), to represent the amount and stores the result in the string_type object val. It returns an iterator designating the first element beyond the monetary input field. Otherwise, the function stores an empty sequence in val and sets ios_base::failbit in st. It returns an iterator designating the first element beyond any prefix of a valid monetary input field. In either case, if the return value equals last, the function sets ios_base::eofbit in st.

The second virtual protected member function behaves the same as the first, except that if successful it converts the optionally-signed digit sequence to a value of type *long double* and stores that value in val.

The format of a monetary input field is determined by the locale facet fac returned by the (effective) call use_facet <moneypunct<E, intl> >(x. getloc()). Specifically:

- fac.neg_format() determines the order in which components of the field occur.
- fac.curr_symbol() determines the sequence of elements that constitutes a currency symbol.
- fac.positive_sign() determines the sequence of elements that constitutes a positive sign.
- fac.negative_sign() determines the sequence of elements that constitutes a negative sign.
- fac.grouping() determines how digits are grouped to the left of any decimal point.
- fac.thousands_sep() determines the element that separates groups of digits to the left of any decimal point.
- fac.decimal_point() determines the element that separates the integer digits from the fraction digits.
- fac.frac_digits() determines the number of significant fraction digits to the right of any decimal point.

If the sign string (fac.negative_sign or fac.positive_sign) has more than one element, only the first element is matched where the element equal to **money_base::sign** appears in the format pattern (fac.neg_format). Any remaining elements are matched at the end of the monetary input field. If neither string has a first element that matches the next element in the monetary input field, the sign string is taken as empty and the sign is positive.

If x.flags() & showbase is nonzero, the string fac.curr_symbol *must* match where the element equal to **money_base::symbol** appears in the format pattern. Otherwise, if money_base::symbol occurs at the end of the format pattern, and if no elements of the sign string remain to be matched, the currency symbol is *not* matched. Otherwise, the currency symbol is *optionally* matched.

If no instances of fac.thousands_sep() occur in the value portion of the monetary input field (where the element equal to **money_base::value** appears in the format pattern), no grouping constraint is imposed. Otherwise, any grouping constraints imposed by fac.grouping() is enforced. Note that the resulting digit sequence represents an integer whose low-order fac.frac_digits() decimal digits are considered to the right of the decimal point.

Arbitrary white space is matched where the element equal to **money_base::space** appears in the format pattern, if it appears other than at the end of the format pattern. Otherwise, no internal white space is matched. An element `c` is considered white space if `use_facet <ctype<E> >(x. getloc()). is(ctype_base:: space, c)` is true.

money_get::get:

```
iter_type get(iter_type first, iter_type last,
    bool intl, ios_base& x, ios_base::iostate& st,
        long double& val) const;
iter_type get(iter_type first, iter_type last,
    bool intl, ios_base& x, ios_base::iostate& st,
        string_type& val) const;
```

Both member functions return `do_get(first, last, intl, x, st, val)`.

money_put

Description: The template class describes an object that can serve as a locale facet, to control conversions of monetary values to sequences of type `E`.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

Synopsis:

```
template<class E,
    class OutIt = ostreambuf_iterator<E> >
    class money_put : public locale::facet {
public:
    typedef E char_type;
    typedef OutIt iter_type;
    typedef basic_string<E> string_type;
    explicit money_put(size_t refs = 0);
    iter_type put(iter_type next, bool intl, ios_base& x,
        E fill, long double& val) const;
    iter_type put(iter_type next, bool intl, ios_base& x,
        E fill, string_type& val) const;
    static locale::id id;
protected:
    ~money_put();
    virtual iter_type do_put(iter_type next, bool intl,
        ios_base& x, E fill, string_type& val) const;
    virtual iter_type do_put(iter_type next, bool intl,
        ios_base& x, E fill, long double& val) const;
    };
```

Constructor:

money_put::money_put:

```
explicit money_put(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

Types:

money_put::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

money_put::iter_type:

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter `OutIt`.

money_put::string_type:

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class basic_string whose objects can store sequences of elements from the source sequence.

Member functions:

money_put::do_put:

```
virtual iter_type do_put(iter_type next, bool intl,
    ios_base& x, E fill, string_type& val) const;
virtual iter_type do_put(iter_type next, bool intl,
    ios_base& x, E fill, long double& val) const;
```

The first virtual protected member function generates sequential elements beginning at `next` to produce a **monetary output field** from the string_type object `val`. The sequence controlled by `val` must begin with one or more decimal digits, optionally preceded by a minus sign (**-**), which represents the amount. The function returns an iterator designating the first element beyond the generated monetary output field.

The second virtual protected member function behaves the same as the first, except that it effectively first converts `val` to a sequence of decimal digits, optionally preceded by a minus sign, then converts that sequence as above.

The format of a monetary output field is determined by the locale facet `fac` returned by the (effective) call `use_facet <moneypunct<E, intl> >(x. getloc())`. Specifically:

- `fac.pos_format()` determines the order in which components of the field are generated for a non-negative value.
- `fac.neg_format()` determines the order in which components of the field are generated for a negative value.
- `fac.curr_symbol()` determines the sequence of elements to generate for a currency symbol.
- `fac.positive_sign()` determines the sequence of elements to generate for a positive sign.
- `fac.negative_sign()` determines the sequence of elements to generate for a negative sign.
- `fac.grouping()` determines how digits are grouped to the left of any decimal point.
- `fac.thousands_sep()` determines the element that separates groups of digits to the left of any decimal point.
- `fac.decimal_point()` determines the element that separates the integer digits from any fraction digits.
- `fac.frac_digits()` determines the number of significant fraction digits to the right of any decimal point.

If the sign string (`fac.negative_sign` or `fac.positive_sign`) has more than one element, only the first element is generated where the element equal to

money_base::sign appears in the format pattern (`fac.neg_format` or `fac.pos_format`). Any remaining elements are generated at the end of the monetary output field.

If `x.flags() & showbase` is nonzero, the string `fac.curr_symbol` is generated where the element equal to **money_base::symbol** appears in the format pattern. Otherwise, no currency symbol is generated.

If no grouping constraints are imposed by `fac.grouping()` (its first element has the value CHAR_MAX) then no instances of `fac.thousands_sep()` are generated in the value portion of the monetary output field (where the element equal to **money_base::value** appears in the format pattern). If `fac.frac_digits()` is zero, then no instance of `fac.decimal_point()` is generated after the decimal digits. Otherwise, the resulting monetary output field places the low-order `fac.frac_digits()` decimal digits to the right of the decimal point.

Padding occurs as for any numeric output field, except that if `x.flags() & x.internal` is nonzero, any internal padding is generated where the element equal to **money_base::space** appears in the format pattern, if it does appear. Otherwise, internal padding occurs before the generated sequence. The padding character is `fill`.

The function calls `x.width(0)` to reset the field width to zero.

money_put::put:

```
iter_type put(iter_type next, bool intl, ios_base& x,
    E fill, long double& val) const;
iter_type put(iter_type next, bool intl, ios_base& x,
    E fill, string_type& val) const;
```

Both member functions return `do_put(next, intl, x, fill, val)`.

moneypunct

Description: The template class describes an object that can serve as a locale facet, to describe the sequences of type `E` used to represent a **monetary input field** or a **monetary output field**. If the template parameter `Intl` is true, international conventions are observed.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

The const static object **intl** stores the value of the template parameter `Intl`.

Synopsis:

```
template<class E, bool Intl>
    class moneypunct
        : public locale::facet, public money_base {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
    explicit moneypunct(size_t refs = 0);
    E decimal_point() const;
    E thousands_sep() const;
    string grouping() const;
    string_type curr_symbol() const;
    string_type positive_sign() const;
    string_type negative_sign() const;
    int frac_digits() const;
```

```
    pattern pos_format(  oonst;
    pattern neg_format() const;
    static const bool intl = Intl;
    static locale::id id;
protected:
    ~moneypunct();
    virtual E do_decimal_point() const;
    virtual E do_thousands_sep() const;
    virtual string do_grouping() const;
    virtual string_type do_curr_symbol() const;
    virtual string_type do_positive_sign() const;
    virtual string_type do_negative_sign() const;
    virtual int do_frac_digits() const;
    virtual pattern do_pos_format() const;
    virtual pattern do_neg_format() const;
    };
```

Constructor:

moneypunct::moneypunct:

```
explicit moneypunct(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

Types:

moneypunct::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

moneypunct::string_type:

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class basic_string whose objects can store copies of the punctuation sequences.

Member functions:

moneypunct::curr_symbol:

```
string_type curr_symbol() const;
```

The member function returns `do_curr_symbol()`.

moneypunct::decimal_point:

```
E decimal_point() const;
```

The member function returns `do_decimal_point()`.

moneypunct::do_curr_symbol:

```
string_type do_curr_symbol() const;
```

The protected virtual member function returns a locale-specific sequence of elements to use as a currency symbol.

moneypunct::do_decimal_point:

```
E do_decimal_point() const;
```

The protected virtual member function returns a locale-specific element to use as a decimal-point.

moneypunct::do_frac_digits:

```
int do_frac_digits() const;
```

The protected virtual member function returns a locale-specific count of the number of digits to display to the right of any decimal point.

moneypunct::do_grouping:

```
string do_grouping() const;
```

The protected virtual member function returns a locale-specific rule for determining how digits are grouped to the left of any decimal point. The encoding is the same as for `lconv::grouping`.

moneypunct::do_neg_format:

```
pattern do_neg_format() const;
```

The protected virtual member function returns a locale-specific rule for determining how to generate a monetary output field for a negative amount. Each of the four elements of `pattern::field` can have the values:

- **none** to match zero or more spaces or generate nothing
- **sign** to match or generate a positive or negative sign
- **space** to match zero or more spaces or generate a space
- **symbol** to match or generate a currency symbol
- **value** to match or generate a monetary value

Components of a monetary output field are generated (and components of a monetary input field are matched) in the order in which these elements appear in `pattern::field`. Each of the values `sign`, `symbol`, `value`, and either `none` or `space` must appear exactly once. The value `none` must not appear first. The value `space` must not appear first or last. If `Intl` is true, the order is `symbol`, `sign`, `none`, then `value`.

The template version of `moneypunct<E, Intl>` returns `{money_base::symbol, money_base::sign, money_base::value, money_base::none}`.

moneypunct::do_negative_sign:

```
string_type do_negative_sign() const;
```

The protected virtual member function returns a locale-specific sequence of elements to use as a negative sign.

moneypunct::do_pos_format:

```
pattern do_pos_format() const;
```

The protected virtual member function returns a locale-specific rule for determining how to generate a monetary output field for a positive amount. (It also determines how to match the components of a monetary input field.) The encoding is the same as for do_neg_format.

The template version of `moneypunct<E, Intl>` returns `{money_base::symbol, money_base::sign, money_base::value, money_base::none}`.

moneypunct::do_positive_sign:

```
string_type do_positive_sign() const;
```

The protected virtual member function returns a locale-specific sequence of elements to use as a positive sign.

moneypunct::do_thousands_sep:

```
E do_thousands_sep() const;
```

The protected virtual member function returns a locale-specific element to use as a group separator to the left of any decimal point.

moneypunct::frac_digits:

```
int frac_digits() const;
```

The member function returns `do_frac_digits()`.

moneypunct::grouping:

```
string grouping() const;
```

The member function returns `do_grouping()`.

moneypunct::neg_format:

```
pattern neg_format() const;
```

The member function returns `do_neg_format()`.

moneypunct::negative_sign:

```
string_type negative_sign() const;
```

The member function returns `do_negative_sign()`.

moneypunct::pos_format:

```
pattern pos_format() const;
```

The member function returns `do_pos_format()`.

moneypunct::positive_sign:

```
string_type positive_sign() const;
```

The member function returns `do_positive_sign()`.

moneypunct::thousands_sep:

```
E thousands_sep() const;
```

The member function returns `do_thousands_sep()`.

moneypunct_byname

```
template<class E, bool Intl>
    class moneypunct_byname
        : public moneypunct<E, Intl> {
public:
    explicit moneypunct_byname(const char *s,
        size_t refs = 0);
protected:
    ~moneypunct_byname();
    };
```

The template class describes an object that can serve as a `locale facet` of type `moneypunct<E, Intl>`. Its behavior is determined by the named locale `s`. The constructor initializes its base object with `moneypunct<E, Intl>(refs)`.

num_get

Description: The template class describes an object that can serve as a locale facet, to control conversions of sequences of type `E` to numeric values.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

Synopsis:

```
template<class E, class InIt = istreambuf_iterator<E> >
    class num_get : public locale::facet {
public:
    typedef E char_type;
    typedef InIt iter_type;
    explicit num_get(size_t refs = 0);
    iter_type get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st,
            long& val) const;
    iter_type get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st,
            unsigned long& val) const;
    iter_type get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st,
            double& val) const;
    iter_type get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st,
            long double& val) const;
    iter_type get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st,
            void *& val) const;
    iter_type get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st,
            bool& val) const;
    static locale::id id;
protected:
    ~num_get();
    virtual iter_type
        do_get(iter_type first, iter_type last,
            ios_base& x, ios_base::iostate& st,
                long& val) const;
    virtual iter_type
        do_get(iter_type first, iter_type last,
            ios_base& x, ios_base::iostate& st,
                unsigned long& val) const;
    virtual iter_type
        do_get(iter_type first, iter_type last,
            ios_base& x, ios_base::iostate& st,
                double& val) const;
    virtual iter_type
        do_get(iter_type first, iter_type last,
            ios_base& x, ios_base::iostate& st,
                long double& val) const;
    virtual iter_type
        do_get(iter_type first, iter_type last,
            ios_base& x, ios_base::iostate& st,
                void *& val) const;
    virtual iter_type
        do_get(iter_type first, iter_type last,
            ios_base& x, ios_base::iostate& st,
                bool& val) const;
    };
```

Constructor:

num_get::num_get:

```
explicit num_get(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

Types:

num_get::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

num_get::iter_type:

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter `InIt`.

Member functions:

num_get::do_get:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        long& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        unsigned long& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        double& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        long double& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        void *& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        bool& val) const;
```

The first virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **integer input field**. If successful, it converts this field to its equivalent value as type *long,* and stores the result in `val`. It returns an iterator designating the first element beyond the numeric input field. Otherwise, the function stores nothing in `val` and sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid integer input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `st`.

The integer input field is converted by the same rules used by the scan functions for matching and converting a series of *char* elements from a file. (Each such *char* element is assumed to map to an equivalent element of type `E` by a simple, one-to-one, mapping.) The equivalent scan conversion specification is determined as follows:

- If `x.flags() & ios_base::basefield == ios_base::oct`, the conversion specification is `lo`.

- If `x.flags() & ios_base::basefield == ios_base::hex`, the conversion specification is `lx`.
- If `x.flags() & ios_base::basefield == 0`, the conversion specification is `li`.
- Otherwise, the conversion specification is `ld`.

The format of an integer input field is further determined by the locale facet `fac` returned by the call `use_facet <numpunct<E>(x. getloc())`. Specifically:

- `fac.grouping()` determines how digits are grouped to the left of any decimal point
- `fac.thousands_sep()` determines the sequence that separates groups of digits to the left of any decimal point

If no instances of `fac.thousands_sep()` occur in the numeric input field, no grouping constraint is imposed. Otherwise, any grouping constraints imposed by `fac.grouping()` is enforced and separators are removed before the scan conversion occurs.

The second virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        unsigned long& val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lu`. If successful it converts the numeric input field to a value of type *unsigned long* and stores that value in `val`.

The third virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        double& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty **floating-point input field**. `fac.decimal_point()` determines the sequence that separates the integer digits from the fraction digits. The equivalent scan conversion specifier is `lf`.

The fourth virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        long double& val) const;
```

behaves the same the third, except that the equivalent scan conversion specifier is `Lf`.

The fifth virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        void *& val) const;
```

behaves the same the first, except that the equivalent scan conversion specifier is `p`.

The sixth virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        bool& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty **boolean input field**. If successful it converts the boolean input field to a value of type `bool` and stores that value in `val`.

A boolean input field takes one of two forms. If `x.flags() & ios_base::boolalpha` is false, it is the same as an integer input field, except that the converted value must be either 0 (for false) or 1 (for true). Otherwise, the sequence must match either `fac.falsename()` (for false), or `fac.truename()` (for true).

num_get::get:

```
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        long& val) const;
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        unsigned long& val) const;
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        double& val) const;
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        long double& val) const;
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        void *& val) const;
iter_type get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st,
        bool& val) const;
```

All member functions return `do_get(first, last, x, st, val)`.

num_put

Description: The template class describes an object that can serve as a locale facet, to control conversions of numeric values to sequences of type `E`.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

Synopsis:

```
template<class E, class OutIt = ostreambuf_iterator<E> >
    class num_put : public locale::facet {
public:
    typedef E char_type;
    typedef OutIt iter_type;
    explicit num_put(size_t refs = 0);
    iter_type put(iter_type next, ios_base& x,
        E fill, long val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, unsigned long val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, double val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, long double val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, const void *val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, bool val) const;
    static locale::id id;
protected:
    ~num_put();
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, long val) const;
```

```
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, unsigned long val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, double val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, long double val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, const void *val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, bool val) const;
    };
```

Constructor:

num_put::num_put:

```
explicit num_put(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

Types:

num_put::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

num_put::iter_type:

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter `OutIt`.

Member functions:

num_put::do_put:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, long val) const;
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, unsigned long val) const;
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, double val) const;
virtual iter_type do_put(iter_type nextp ios_base& x,
    E fill, long double val) const;
virtual iter_type do_put(iter_type nextp ios_base& x,
    E fill, const void *val) const;
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, bool val) const;
```

The first virtual protected member function generates sequential elements beginning at `next` to produce an **integer output field** from the value of `val`. The function returns an iterator designating the next place to insert an element beyond the generated integer output field.

The integer output field is generated by the same rules used by the print functions for generating a series of *char* elements to a file. (Each such *char* element is assumed to map to an equivalent element of type `E` by a simple, one-to-one, mapping.) Where a print function pads a field with either spaces or the digit `0`, however, `do_put` instead uses `fill`. The equivalent print conversion specification is determined as follows:

- If x.flags() & ios_base::basefield == ios_base::oct, the conversion specification is lo.
- If x.flags() & ios_base::basefield == ios_base::hex, the conversion specification is lx.
- Otherwise, the conversion specification is ld.

If x.width() is nonzero, a field width of this value is prepended. The function then calls x.width(0) to reset the field width to zero.

Padding occurs only if the minimum number of elements N required to specify the output field is less than x.width(). Such padding consists of a sequence of N - width() copies of fill. Padding then occurs as follows:
- If x.flags() & ios_base::adjustfield == ios_base::left, the flag - is prepended. (Padding occurs after the generated text.)
- If x.flags() & ios_base::adjustfield == ios_base::internal, the flag 0 is prepended. (For a numeric output field, padding occurs where the print functions pad with 0.)
- Otherwise, no additional flag is prepended. (Padding occurs before the generated sequence.)

Finally:
- If x.flags() & ios_base::showpos is nonzero, the flag + is prepended to the conversion specification.
- If x.flags() & ios_base::showbase is nonzero, the flag # is prepended to the conversion specification.

The format of an integer output field is further determined by the locale facet fac returned by the call use_facet <numpunct<E>(x. getloc()). Specifically:
- fac.grouping() determines how digits are grouped to the left of any decimal point
- fac.thousands_sep() determines the sequence that separates groups of digits to the left of any decimal point

If no grouping constraints are imposed by fac.grouping() (its first element has the value CHAR_MAX) then no instances of fac.thousands_sep() are generated in the output field. Otherwise, separators are inserted after the print conversion occurs.

The second virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, unsigned long val) const;
```

behaves the same as the first, except that it replaces a conversion specification of ld with lu.

The third virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, double val) const;
```

behaves the same as the first, except that it produces a **floating-point output field** from the value of val. fac.decimal_point() determines the sequence that separates the integer digits from the fraction digits. The equivalent print conversion specification is determined as follows:

- If `x.flags() & ios_base::floatfield == ios_base::fixed`, the conversion specification is `lf`.
- If `x.flags() & ios_base::floatfield == ios_base::scientific`, the conversion specification is `le`. If `x.flags() & ios_base::uppercase` is nonzero, `e` is replaced with `E`.
- Otherwise, the conversion specification is `lg`. If `x.flags() & ios_base::uppercase` is nonzero, `g` is replaced with `G`.

If `x.flags() & ios_base::fixed` is nonzero, or if `x.precision()` is greater than zero, a precision with the value `x.precision()` is prepended to the conversion specification. Any padding behaves the same as for an integer output field. The padding character is `fill`. Finally:

- If `x.flags() & ios_base::showpos` is nonzero, the flag `+` is prepended to the conversion specification.
- If `x.flags() & ios_base::showpoint` is nonzero, the flag `#` is prepended to the conversion specification.

The fourth virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, long double val) const;
```

behaves the same the third, except that the qualifier `l` in the conversion specification is replaced with `L`.

The fifth virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, const void *val) const;
```

behaves the same the first, except that the conversion specification is `p`, plus any qualifier needed to specify padding.

The sixth virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, bool val) const;
```

behaves the same as the first, except that it generates a **boolean output field** from `val`.

A boolean output field takes one of two forms. If `x.flags() & ios_base::boolalpha` is false, the generated sequence is either `0` (for false) or `1` (for true). Otherwise, the generated sequence is either `fac.falsename()` (for false), or `fac.truename()` (for true).

num_put::put:

```
iter_type put(iter_type next, ios_base& x,
    E fill, long val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, unsigned long val) const;
iter_type put(iter_type iter_type next, ios_base& x,
    E fill, double val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, long double val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, const void *val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, bool val) const;
```

All member functions return `do_put(next, x, fill, val)`.

numpunct

Description: The template class describes an object that can serve as a locale facet, to describe the sequences of type `E` used to represent the input fields matched by num_get or the output fields generated by num_get.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

Synopsis:

```
template<class E, class numpunct : public locale::facet {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
    explicit numpunct(size_t refs = 0);
    E decimal_point() const;
    E thousands_sep() const;
    string grouping() const;
    string_type truename() const;
    string_type falsename() const;
    static locale::id id;
protected:
    ~numpunct();
    virtual E do_decimal_point() const;
    virtual E do_thousands_sep() const;
    virtual string do_grouping() const;
    virtual string_type do_truename() const;
    virtual string_type do_falsename() const;
    };
```

Constructor:

numpunct::numpunct:

```
explicit numpunct(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

Types:

numpunct::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

numpunct::string_type:

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class basic_string whose objects can store copies of the punctuation sequences.

Member functions:

numpunct::decimal_point:

```
E decimal_point() const;
```

The member function returns `do_decimal_point()`.

numpunct::do_decimal_point:

```
E do_decimal_point() const;
```

The protected virtual member function returns a locale-specific element to use as a decimal-point.

numpunct::do_falsename:

```
string_type do_falsename() const;
```

The protected virtual member function returns a locale-specific sequence to use as a text representation of the value false.

numpunct::do_grouping:

```
string do_grouping() const;
```

The protected virtual member function returns a locale-specific rule for determining how digits are grouped to the left of any decimal point. The encoding is the same as for `lconv::grouping`.

numpunct::do_thousands_sep:

```
E do_thousands_sep() const;
```

The protected virtual member function returns a locale-specific element to use as a group separator to the left of any decimal point.

numpunct::do_truename:

```
string_type do_truename() const;
```

The protected virtual member function returns a locale-specific sequence to use as a text representation of the value true.

numpunct::falsename:

```
string_type falsename() const;
```

The member function returns `do_falsename()`.

numpunct::grouping:

```
string grouping() const;
```

The member function returns `do_grouping()`.

numpunct::thousands_sep:

```
E thousands_sep() const;
```

The member function returns `do_thousands_sep()`.

numpunct::truename:

```
string_type falsename() const;
```

The member function returns `do_truename()`.

numpunct_byname

Description: The template class describes an object that can serve as a `locale facet` of type `numpunct<E>`. Its behavior is determined by the named locale `s`. The constructor initializes its base object with `numpunct<E>(refs)`.

Synopsis:

```
template<class E>
    class numpunct_byname : public numpunct<E> {
public:
    explicit numpunct_byname(const char *s,
        size_t refs = 0);
protected:
    ~numpunct_byname();
    };
```

time_base

Description: The class serves as a base class for facets of template class time_get. It defines just the enumerated type **dateorder** and several constants of this type. Each of the constants characterizes a different way to order the components of a date. The constants are:

- **no_order** specifies no particular order.
- **dmy** specifies the order day, month, then year, as in `2 December 1979`.
- **mdy** specifies the order month, day, then year, as in `December 2, 1979`.
- **ymd** specifies the order year, month, then day, as in `1979/12/2`.
- **ydm** specifies the order year, day, then month, as in `1979: 2 Dec`.

Synopsis:

```
class time_base {
public:
    enum dateorder {no_order, dmy, mdy, ymd, ydm};
    };
```

time_get

Description: The template class describes an object that can serve as a locale facet, to control conversions of sequences of type `E` to time values.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

Synopsis:

```
template<class E, class InIt = istreambuf_iterator<E> >
    class time_get : public locale::facet {
public:
    typedef E char_type;
    typedef InIt iter_type;
    explicit time_get(size_t refs = 0);
    dateorder date_order() const;
    iter_type get_time(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_date(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_weekday(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_month(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_year(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
```

```
    static locale::id id;
protected:
    ~time_get();
    virtual dateorder do_date_order() const;
    virtual iter_type
        do_get_time(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    virtual iter_type
        do_get_date(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    virtual iter_type
        do_get_weekday(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    virtual iter_type
        do_get_month(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    virtual iter_type
        do_get_year(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    };
```

Constructor:

time_get::time_get:

```
explicit time_get(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

Types:

time_get::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

time_get::iter_type:

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter `InIt`.

Member functions:

time_get::date_order:

```
dateorder date_order() const;
```

The member function returns `date_order()`.

time_get::do_date_order:

```
virtual dateorder do_date_order() const;
```

The virtual protected member function returns a value of type `time_base::dateorder`, which describes the order in which date components are matched by do_get_date. In this implementation, the value is `time_base::mdy`, corresponding to dates of the form `December 2, 1979.`

time_get::do_get_date:

```
virtual iter_type
    do_get_date(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **date input field**. If successful, it converts this field to its equivalent value as the components `tm::tm_mon`, `tm::tm_day`, and `tm::tm_year`, and stores the results in `pt->tm_mon`, `pt->tm_day` and `pt->tm_year`, respectively. It returns an iterator designating the first element beyond the date input field. Otherwise, the function sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid date input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `st`.

In this implementation, the date input field has the form `MMM DD, YYYY`, where:
- `MMM` is matched by calling get_month, giving the month.
- `DD` is a sequence of decimal digits whose corresponding numeric value must be in the range [1, 31], giving the day of the month.
- `YYYY` is matched by calling get_year, giving the year.
- The literal spaces and commas must match corresponding elements in the input sequence.

time_get::do_get_month:

```
virtual iter_type
    do_get_month(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **month input field**. If successful, it converts this field to its equivalent value as the component `tm::tm_mon`, and stores the result in `pt->tm_mon`. It returns an iterator designating the first element beyond the month input field. Otherwise, the function sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid month input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `st`.

The month input field is a sequence that matches the longest of a set of locale-specific sequences, such as: `Jan`, `January`, `Feb`, `February`, etc. The converted value is the number of months since January.

time_get::do_get_time:

```
virtual iter_type
    do_get_time(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **time input field**. If successful, it converts this field to its equivalent value as the components `tm::tm_hour`, `tm::tm_min`, and `tm::tm_sec`, and stores the results in `pt->tm_hour`, `pt->tm_min` and `pt->tm_sec`, respectively. It returns an iterator designating the first element beyond the time input field. Otherwise, the function sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid time input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `st`.

In this implementation, the time input field has the form `HH:MM:SS`, where:

- HH is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 24), giving the hour of the day.
- MM is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 60), giving the minutes past the hour.
- SS is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 60), giving the seconds past the minute.
- The literal colons must match corresponding elements in the input sequence.

time_get::do_get_weekday:

```
virtual iter_type
    do_get_weekday(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **weekday input field**. If successful, it converts this field to its equivalent value as the component `tm::tm_wday`, and stores the result in `pt->tm_wday`. It returns an iterator designating the first element beyond the weekday input field. Otherwise, the function sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid weekday input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `st`.

The weekday input field is a sequence that matches the longest of a set of locale-specific sequences, such as: `Sun`, `Sunday`, `Mon`, `Monday`, etc. The converted value is the number of days since Sunday.

time_get::do_get_year:

```
virtual iter_type
    do_get_year(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **year input field**. If successful, it converts this field to its equivalent value as the component `tm::tm_year`, and stores the result in `pt->tm_year`. It returns an iterator designating the first element beyond the year input field. Otherwise, the function sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid year input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `st`.

The year input field is a sequence of decimal digits whose corresponding numeric value must be in the range [1900, 2036). The stored value is this value minus 1900. In this implementation, a numeric value in the range [0, 136) is also permissible. It is stored unchanged.

time_get::get_date:

```
iter_type get_date(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns `do_get_date(first, last, x, st, pt)`.

time_get::get_month:

```
iter_type get_month(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns do_get_month(first, last, x, st, pt).

time_get::get_time:

```
iter_type get_time(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns do_get_time(first, last, x, st, pt).

time_get::get_weekday:

```
iter_type get_weekday(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns do_get_weekday(first, last, x, st, pt).

time_get::get_year:

```
iter_type get_year(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns do_get_year(first, last, x, st, pt).

time_get_byname

Description: The template class describes an object that can serve as a locale facet of type time_get<E, InIt>. Its behavior is determined by the named locale s. The constructor initializes its base object with time_get<E, InIt>(refs).

Synopsis:

```
template<class E, class InIt>
    class time_get_byname : public time_get<E, InIt> {
public:
    explicit time_get_byname(const char *s,
        size_t refs = 0);
protected:
    ~time_get_byname();
    };
```

time_put

Description: The template class describes an object that can serve as a locale facet, to control conversions of time values to sequences of type E.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in id.

Synopsis:

```
template<class E, class OutIt = ostreambuf_iterator<E> >
    class time_put : public locale::facet {
public:
    typedef E char_type;
    typedef OutIt iter_type;
    explicit time_put(size_t refs = 0);
    iter_type put(iter_type next, ios_base& x,
        char_type fill, const tm *pt, char fmt, char mod = 0) const;
    iter_type put(iter_type next, ios_base& x,
        char_type fill, const tm *pt, const E *first, const E *last) const;
    static locale::id id;
protected:
```

```
~time_put();
virtual iter_type do_put(iter_type next, ios_base& x,
    char_type fill, const tm *pt, char fmt, char mod = 0) const;
};
```

Constructor:

time_put::time_put:

```
explicit time_put(size_t refs = 0);
```

The constructor initializes its base object with locale::facet(refs).

Types:

time_put::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

time_put::iter_type:

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter OutIt.

Member functions:

time_put::do_put:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    char_type fill, const tm *pt, char fmt, char mod = 0) const;
```

The virtual protected member function generates sequential elements beginning at next from time values stored in the object *pt, of type tm. The function returns an iterator designating the next place to insert an element beyond the generated output.

The output is generated by the same rules used by strftime, with a last argument of pt, for generating a series of *char* elements into an array. (Each such *char* element is assumed to map to an equivalent element of type E by a simple, one-to-one, mapping.) If mod equals zero, the effective format is "%F", where F equals fmt. Otherwise, the effective format is "%MF", where M equals mod.

The parameter fill is not used.

time_put::put:

```
iter_type put(iter_type next, ios_base& x,
    char_type fill, const tm *pt, char fmt, char mod = 0) const;
iter_type put(iter_type next, ios_base& x,
    char_type fill, const tm *pt, const E *first, const E *last) const;
```

The first member function returns do_put(next, x, fill, pt, fmt, mod). The second member function copies to *next++ any element in the interval [first, last) other than a percent (%). For a percent followed by a character C in the interval [first, last), the function instead evaluates next = do_put(next, x, fill, pt, C, 0) and skips past C. If, however, C is a qualifier character from the set EOQ#, followed by a character C2 in the interval [first, last), the function instead evaluates next = do_put(next, x, fill, pt, C2, C) and skips past C2.

time_put_byname

Description: The template class describes an object that can serve as a `locale facet` of type `time_put<E, OutIt>`. Its behavior is determined by the named locale `s`. The constructor initializes its base object with `time_put<E, OutIt>(refs)`.

Synopsis:

```
template<class E, class OutIt>
    class time_put_byname : public time_put<E, OutIt> {
public:
    explicit time_put_byname(const char *s,
        size_t refs = 0);
protected:
    ~time_put_byname();
    };
```

Template functions

has_facet

```
template<class Facet>
    bool has_facet(const locale& loc);
```

The template function returns true if a locale facet of class `Facet` is listed within the locale object `loc`.

isalpha

```
template<class E>
    bool isalpha(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: alpha, c)`.

iscntrl

```
template<class E>
    bool iscntrl(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: cntrl, c)`.

isdigit

```
template<class E>
    bool isdigit(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: digit, c)`.

isgraph

```
template<class E>
    bool isgraph(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: graph, c)`.

islower

```
template<class E>
    bool islower(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: lower, c)`.

isprint

```
template<class E>
    bool isprint(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: print, c)`.

ispunct

```
template<class E>
    bool ispunct(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: punct, c)`.

isspace

```
template<class E>
    bool isspace(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: space, c)`.

isupper

```
template<class E>
    bool isupper(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: upper, c)`.

isxdigit

```
template<class E>
    bool isxdigit(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: xdigit, c)`.

tolower

```
template<class E>
    E tolower(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). tolower(c)`.

toupper

```
template<class E>
    E toupper(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). toupper(c)`.

use_facet

```
template<class Facet>
    const Facet& use_facet(const locale& loc);
```

The template function returns a reference to the `locale facet` of class `Facet` listed within the locale object `loc`. If no such object is listed, the function throws an object of class bad_cast.

<map>

Description

Include the STL standard header **<map>** to define the container template classes `map` and `multimap`, and their supporting templates.

Synopsis

```
namespace std {
template<class Key, class T, class Pred, class A>
    class map;
template<class Key, class T, class Pred, class A>
    class multimap;

        // TEMPLATE FUNCTIONS
template<class Key, class T, class Pred, class A>
    bool operator==(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator==(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    void swap(
```

```
        map<Key, T, Pred, A>& lhs,
        map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    void swap(
        multimap<Key, T, Pred, A>& lhs,
        multimap<Key, T, Pred, A>& rhs);
    }
```

Macros

__IBM_FAST_SET_MAP_ITERATOR

```
#define __IBM_FAST_SET_MAP_ITERATOR
```

The __IBM_FAST_SET_MAP_ITERATOR macro enables doubly-linked list data structures for the Standard C++ Library set and map classes and can assist with improving performance. By default, this macro must be explicitly specified to gain any potential performance improvement. Any code compiled with this macro cannot be mixed with code compiled without the macro.

Classes

map

Description: The template class describes an object that controls a varying-length sequence of elements of type `pair<const Key, T>`. The sequence is ordered by the predicate `Pred`. The first element of each pair is the **sort key** and the second is its associated **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this stored object by calling the member function `key_comp()`. Such a function object must impose a total ordering on sort keys of type `Key`. For any element `x` that precedes `y` in the sequence, `key_comp()(y.first, x.first)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class multimap, an object of template class `map` ensures that `key_comp()(x.first, y.first)` is true. (Each key is unique.)

The object allocates and frees storage for the sequence it controls through a stored allocator object of class `A`. Such an allocator object must have the same external interface as an object of template class allocator. Note that the stored allocator object is *not* copied when the container object is assigned.

Synopsis:

```
template<class Key, class T, class Pred = less<Key>,
    class A = allocator<pair<const Key, T> > >
    class map {
public:
    typedef Key key_type;
    typedef T mapped_type;
    typedef Pred key_compare;
    typedef A allocator_type;
    typedef pair<const Key, T> value_type;
    class value_compare;
    typedef A::pointer pointer;
    typedef A::const_pointer const_pointer;
```

```
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
    map();
    explicit map(const Pred& comp);
    map(const Pred& comp, const A& al);
    map(const map& x);
    template<class InIt>
        map(InIt first, InIt last);
    template<class InIt>
        map(InIt first, InIt last,
            const Pred& comp);
    template<class InIt>
        map(InIt first, InIt last,
            const Pred& comp, const A& al);
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    mapped_type operator[](const Key& key);
    pair<iterator, bool> insert(const value_type& x);
    iterator insert(iterator it, const value_type& x);
    template<class InIt>
        void insert(InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& key);
    void clear();
    void swap(map& x);
    key_compare key_comp() const;
    value_compare value_comp() const;
    iterator find(const Key& key);
    const_iterator find(const Key& key) const;
    size_type count(const Key& key) const;
    iterator lower_bound(const Key& key);
    const_iterator lower_bound(const Key& key) const;
    iterator upper_bound(const Key& key);
    const_iterator upper_bound(const Key& key) const;
    pair<iterator, iterator> equal_range(const Key& key);
    pair<const_iterator, const_iterator>
        equal_range(const Key& key) const;
    };
```

Constructor:

map::map:

```
map();
explicit map(const Pred& comp);
map(const Pred& comp, const A& al);
map(const map& x);
template<class InIt>
    map(InIt first, InIt last);
```

```
template<class InIt>
    map(InIt first, InIt last,
        const Pred& comp);
template<class InIt>
    map(InIt first, InIt last,
        const Pred& comp, const A& al);
```

All constructors store an allocator object and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

All constructors also store a function object that can later be returned by calling `key_comp()`. The function object is the argument `comp`, if present. For the copy constructor, it is `x.key_comp()`). Otherwise, it is `Pred()`.

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by `x`. The last three constructors specify the sequence of element values `[first, last)`.

Types:

map::allocator_type:

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

map::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

map::const_pointer:

```
typedef A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

map::const_reference:

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

map::const_reverse_iterator:

```
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

map::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

map::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

map::key_compare:

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of two elements in the controlled sequence.

map::key_type:

```
typedef Key key_type;
```

The type describes the sort key object stored in each element of the controlled sequence.

map::mapped_type:

```
typedef T mapped_type;
```

The type is a synonym for the template parameter `T`.

map::pointer:

```
typedef A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

map::reference:

```
typedef A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

map::reverse_iterator:

```
typedef reverse_iterator<iterator> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

map::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type `T2`.

map::value_compare:

```
class value_compare
    : public binary_function<value_type, value_type,
        bool> {
public:
    bool operator()(const value_type& x,
        const value_type& y) const
        {return (comp(x.first, y.first)); }
protected:
    value_compare(key_compare pr)
        : comp(pr) {}
    key_compare comp;
    };
```

The type describes a function object that can compare the sort keys in two elements to determine their relative order in the controlled sequence. The function object stores an object **comp** of type key_compare. The member function **operator()** uses this object to compare the sort-key components of two element.

map::value_type:

```
typedef pair<const Key, T> value_type;
```

The type describes an element of the controlled sequence.

Member functions:

map::begin:

```
const_iterator begin() const;
iterator begin();
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

map::clear:

```
void clear();
```

The member function calls `erase( begin(), end())`.

map::count:

```
size_type count(const Key& key) const;
```

The member function returns the number of elements `x` in the range `[lower_bound(key), upper_bound(key))`.

map::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

map::end:

```
const_iterator end() const;
iterator end();
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

map::equal_range:

```
pair<iterator, iterator> equal_range(const Key& key);
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
```

The member function returns a pair of iterators `x` such that `x.first == lower_bound(key)` and `x.second == upper_bound(key)`.

map::erase:

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements in the interval `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member function removes the elements with sort keys in the range `[lower_bound(key), upper_bound(key))`. It returns the number of elements it removes.

The member functions never throw an exception.

map::find:

```
iterator find(const Key& key);
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering to `key`. If no such element exists, the function returns `end()`.

map::get_allocator:

```
A get_allocator() const;
```

The member function returns the stored allocator object.

map::insert:

```
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
```

The first member function determines whether an element `y` exists in the sequence whose key has equivalent ordering to that of `x`. If not, it creates such an element `y` and initializes it with `x`. The function then determines the iterator `it` that designates `y`. If an insertion occurred, the function returns `pair(it, true)`. Otherwise, it returns `pair(it, false)`.

The second member function returns `insert(x)`, using `it` as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows `it`.) The third member function inserts the sequence of element values, for each `it` in the range `[first, last)`, by calling `insert(*it)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

map::key_comp:

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object defines the member function:

```
bool operator()(const Key& x, const Key& y);
```

which returns true if `x` strictly precedes `y` in the sort order.

map::lower_bound:

```
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(x. first, key)` is false.

If no such element exists, the function returns `end()`.

map::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

map::operator[]:

```
T& operator[](const Key& key);
```

The member function determines the iterator `it` as the return value of `insert( value_type(key, T())`. (It inserts an element with the specified key if no such element exists.) It then returns a reference to `(*it).second`.

map::rbegin:

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

map::rend:

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

map::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

map::swap:

```
void swap(map& x);
```

The member function swaps the controlled sequences between `*this` and `x`. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type `Pred`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

map::upper_bound:

```
iterator upper_bound(const Key& key);
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(key, x.first)` is true.

If no such element exists, the function returns `end()`.

map::value_comp:

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

multimap

Description: The template class describes an object that controls a varying-length sequence of elements of type `pair<const Key, T>`. The sequence is ordered by the predicate `Pred`. The first element of each pair is the **sort key** and the second is its associated **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this stored object by calling the member function `key_comp()`. Such a function object must impose a total ordering on sort keys of type `Key`. For any element `x` that precedes `y` in the sequence, `key_comp()(y.first, x.first)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class map, an object of template class `multimap` does not ensure that `key_comp()(x.first, y.first)` is true. (Keys need not be unique.)

The object allocates and frees storage for the sequence it controls through a stored allocator object of class `A`. Such an allocator object must have the same external interface as an object of template class allocator. Note that the stored allocator object is *not* copied when the container object is assigned.

Synopsis:

```
template<class Key, class T, class Pred = less<Key>,
    class A = allocator<pair<const Key, T> > >
    class multimap {
public:
    typedef Key key_type;
    typedef T mapped_type;
    typedef Pred key_compare;
    typedef A allocator_type;
    typedef pair<const Key, T> value_type;
    class value_compare;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
    multimap();
    explicit multimap(const Pred& comp);
    multimap(const Pred& comp, const A& al);
    multimap(const multimap& x);
    template<class InIt>
        multimap(InIt first, InIt last);
    template<class InIt>
        multimap(InIt first, InIt last,
            const Pred& comp);
    template<class InIt>
        multimap(InIt first, InIt last,
            const Pred& comp, const A& al);
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    iterator insert(const value_type& x);
    iterator insert(iterator it, const value_type& x);
    template<class InIt>
        void insert(InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& key);
    void clear();
    void swap(multimap& x);
    key_compare key_comp() const;
    value_compare value_comp() const;
    iterator find(const Key& key);
    const_iterator find(const Key& key) const;
    size_type count(const Key& key) const;
    iterator lower_bound(const Key& key);
    const_iterator lower_bound(const Key& key) const;
    iterator upper_bound(const Key& key);
    const_iterator upper_bound(const Key& key) const;
    pair<iterator, iterator> equal_range(const Key& key);
    pair<const_iterator, const_iterator>
        equal_range(const Key& key) const;
    };
```

Constructor:

multimap::multimap:

```
multimap();
explicit multimap(const Pred& comp);
multimap(const Pred& comp, const A& al);
multimap(const multimap& x);
template<class InIt>
    multimap(InIt first, InIt last);
template<class InIt>
    multimap(InIt first, InIt last,
        const Pred& comp);
template<class InIt>
    multimap(InIt first, InIt last,
        const Pred& comp, const A& al);
```

All constructors store an allocator object and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator ()`. Otherwise, it is `A()`.

All constructors also store a function object that can later be returned by calling `key_comp()`. The function object is the argument `comp`, if present. For the copy constructor, it is `x.key_comp ())`. Otherwise, it is `Pred()`.

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by `x`. The last three constructors specify the sequence of element values `[first, last)`.

Types:

multimap::allocator_type:

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

multimap::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

multimap::const_pointer:

```
typedef A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

multimap::const_reference:

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

multimap::const_reverse_iterator:

```
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

multimap::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

multimap::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T0.

multimap::key_compare:

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of two elements in the controlled sequence.

multimap::key_type:

```
typedef Key key_type;
```

The type describes the sort key object stored in each element of the controlled sequence.

multimap::mapped_type:

```
typedef T mapped_type;
```

The type is a synonym for the template parameter T.

multimap::pointer:

```
typedef A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

multimap::reference:

```
typedef A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

multimap::reverse_iterator:

```
typedef reverse_iterator<iterator> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

multimap::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

multimap::value_compare:

```
class value_compare
    : public binary_function<value_type, value_type,
        bool> {
public:
    bool operator()(const value_type& x,
        const value_type& y) const
        {return (comp(x.first, y.first)); }
protected:
    value_compare(key_compare pr)
        : comp(pr) {}
    key_compare comp;
    };
```

The type describes a function object that can compare the sort keys in two elements to determine their relative order in the controlled sequence. The function object stores an object **comp** of type key_compare. The member function **operator()** uses this object to compare the sort-key components of two element.

multimap::value_type:

```
typedef pair<const Key, T> value_type;
```

The type describes an element of the controlled sequence.

Member functions:

multimap::begin:

```
const_iterator begin() const;
iterator begin();
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

multimap::clear:

```
void clear();
```

The member function calls erase(begin(), end()).

multimap::count:

```
size_type count(const Key& key) const;
```

The member function returns the number of elements x in the range [lower_bound(key), upper_bound(key)).

multimap::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

multimap::end:

```
const_iterator end() const;
iterator end();
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

multimap::equal_range:

```
pair<iterator, iterator> equal_range(const Key& key);
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
```

The member function returns a pair of iterators `x` such that `x.first == lower_bound(key)` and `x.second == upper_bound(key)`.

multimap::erase:

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes the elements with sort keys in the range `[lower_bound(key), upper_bound(key))`. It returns the number of elements it removes.

The member functions never throw an exception.

multimap::find:

```
iterator find(const Key& key);
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering to `key`. If no such element exists, the function returns `end()`.

multimap::get_allocator:

```
A get_allocator() const;
```

The member function returns the stored allocator object.

multimap::insert:

```
iterator insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
```

The first member function inserts the element `x` in the controlled sequence, then returns the iterator that designates the inserted element. The second member function returns `insert(x)`, using `it` as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows `it`.) The third member function inserts the sequence of element values, for each `it` in the range `[first, last)`, by calling `insert(*it)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

multimap::key_comp:

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object defines the member function:

```
bool operator()(const Key& x, const Key& y);
```

which returns true if `x` strictly precedes `y` in the sort order.

multimap::lower_bound:

```
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(x. first, key)` is false.

If no such element exists, the function returns `end()`.

multimap::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

multimap::rbegin:

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

multimap::rend:

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

multimap::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

multimap::swap:

```
void swap(multimap& x);
```

The member function swaps the controlled sequences between `*this` and `x`. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type Pred, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

multimap::upper_bound:

```
iterator upper_bound(const Key& key);
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(key, x.first)` is true.

If no such element exists, the function returns `end()`.

multimap::value_comp:

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

Template functions

operator!=

```
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class Key, class T, class Pred, class A>
    bool operator==(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator==(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The first template function overloads `operator==` to compare two objects of template class multimap. The second template function overloads `operator==` to compare two objects of template class multimap. Both functions return `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end (), rhs.begin())`.

operator<

```
template<class Key, class T, class Pred, class A>
    bool operator<(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The first template function overloads `operator<` to compare two objects of template class multimap. The second template function overloads `operator<` to compare two objects of template class multimap. Both functions return `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class Key, class T, class Pred, class A>
    bool operator>(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class Key, class T, class Pred, class A>
    bool operator>=(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class Key, class T, class Pred, class A>
    void swap(
        map <Key, T, Pred, A>& lhs,
        map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    void swap(
        multimap <Key, T, Pred, A>& lhs,
        multimap <Key, T, Pred, A>& rhs);
```

The template function executes `lhs.swap (rhs)`.

<memory>

Description

Include the STL standard header **<memory>** to define a class, an operator, and several templates that help allocate and free objects. As well, **<memory>** defines the smart pointer templates auto_ptr, shared_ptr, and weak_ptr to assist in the management of dynamically allocated objects.

Note: Additional functionality has been added to this header for TR1. To enable this functionality, you must define the macro __IBMCPP_TR1__ and use the **TARGET** compiler option to specify a valid release level. Valid release levels for TR1 support are zOSV1R12 or later. For example, you can specify the **TARGET** option as follows: `TARGET(zOSV1R12)`.

Any release prior to zOSV1R12 is invalid for the use of TR1. If TR1 code is used in an application to be compiled on an earlier platform, the compiler will issue the `#error` directive.

Synopsis

```
namespace std {
template<class Ty>
    class allocator;
template<>
    class allocator<void>;
template<class FwdIt, class Ty>
    class raw_storage_iterator;
template<class Ty>
    class auto_ptr;
template<class Ty>
    class auto_ptr_ref;

        // TEMPLATE OPERATORS
template<class Ty>
    bool operator==(const allocator<Ty>& left,
        const allocator<Ty>& right);
template<class Ty>
    bool operator!=(const allocator<Ty>& left,
        const allocator<Ty>& right);

        // TEMPLATE FUNCTIONS
template<class Ty>
    pair<Ty *, ptrdiff_t>
        get_temporary_buffer(ptrdiff_t count);
template<class Ty>
    void return_temporary_buffer(Ty *pbuf);
template<class InIt, class FwdIt>
    FwdIt uninitialized_copy(InIt first, InIt last,
        FwdIt dest);
template<class FwdIt, class Ty>
    void uninitialized_fill(FwdIt first, FwdIt last,
        const Ty& val);
template<class FwdIt, class Size, class Ty>
    void uninitialized_fill_n(FwdIt first, Size count,
        const Ty& val);
    }
```

Classes

allocator

Description: The template class describes an object that manages storage allocation and freeing for arrays of objects of `non-const`, `non-reference`, object type `Ty`. An object of class `allocator` is the default **allocator object** specified in the constructors for several container template classes in the Standard C++ Library.

Template class `allocator` supplies several type definitions that are rather pedestrian. They hardly seem worth defining. But another class with the same members might choose more interesting alternatives. Constructing a container with an allocator object of such a class gives individual control over allocation and freeing of elements controlled by that container.

For example, an allocator object might allocate storage on a **private heap**. Or it might allocate storage on a **far heap**, requiring nonstandard pointers to access the allocated objects. Or it might specify, through the type definitions it supplies, that elements be accessed through special **accessor objects** that manage **shared memory**, or perform automatic **garbage collection**. Hence, a class that allocates storage using an allocator object should use these types religiously for declaring pointer and reference objects (as do the containers in the Standard C++ Library).

Thus, an allocator defines the types (among others):
- pointer — behaves like a pointer to `Ty`
- const_pointer — behaves like a const pointer to `Ty`
- reference — behaves like a reference to `Ty`
- const_reference — behaves like a const reference to `Ty`

These types specify the form that pointers and references must take for allocated elements. (`allocator::pointer` is not necessarily the same as `Ty *` for all allocator objects, even though it has this obvious definition for class `allocator`.)

Synopsis:

```
template<class Ty>
    class allocator {
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef Ty *pointer;
    typedef const Ty *const_pointer;
    typedef Ty& reference;
    typedef const Ty& const_reference;
    typedef Ty value_type;
    pointer address(reference val) const;
    const_pointer address(const_reference val) const;
    template<class Other>
        struct rebind;
    allocator() throw();
    template<class Other>
        allocator(const allocator<Other>& right) throw();
    template<class Other>
        allocator& operator=(const allocator<Other>& right);
    pointer allocate(size_type count,
        typename allocator<void>::const_pointer *hint = 0);
    void deallocate(pointer ptr, size_type count);
    void construct(pointer ptr, const Ty& val);
    void construct(pointer ptr, Ty&& val);
    void destroy(pointer ptr);
    size_type max_size() const throw();
    };
```

Constructor:

allocator::allocator:

```
allocator() throw();
template<class Other>
    allocator(const allocator<Other>& right) throw();
```

The constructor does nothing. In general, however, an allocator object constructed from another allocator object should compare equal to it (and hence permit intermixing of object allocation and freeing between the two allocator objects).

Types:

allocator::const_pointer:

```
typedef const Ty *pointer;
```

The pointer type describes an object `ptr` that can designate, via the expression `*ptr`, any const object that an object of template class `allocator` can allocate.

allocator::const_reference:

```
typedef const Ty& const_reference;
```

The reference type describes an object that can designate any const object that an object of template class `allocator` can allocate.

allocator::difference_type:

```
typedef ptrdiff_t difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in a sequence that an object of template class `allocator` can allocate.

allocator::pointer:

```
typedef Ty *pointer;
```

The pointer type describes an object `ptr` that can designate, via the expression `*ptr`, any object that an object of template class `allocator` can allocate.

allocator::reference:

```
typedef Ty& reference;
```

The reference type describes an object that can designate any object that an object of template class `allocator` can allocate.

allocator::size_type:

```
typedef size_t size_type;
```

The unsigned integer type describes an object that can represent the length of any sequence that an object of template class `allocator` can allocate.

allocator::value_type:

```
typedef Ty value_type;
```

The type is a synonym for the template parameter `Ty`.

Member classes:

allocator::rebind:

```
template<class Other>
    struct rebind {
    typedef allocator<Other> other;
    };
```

The member template class defines the type **other**. Its sole purpose is to provide the type name `allocator<Other>` given the type name `allocator<Ty>`.

For example, given an allocator object `al` of type `A`, you can allocate an object of type `Other` with the expression:

```
A::rebind<Other>::other(al).allocate(1, (Other *)0)
```

Or, you can simply name its pointer type by writing the type:

```
A::rebind<Other>::other::pointer
```

Member functions:

allocator::address:

```
pointer address(reference val) const;
const_pointer address(const_reference x) const;
```

The member functions return the address of `val`, in the form that pointers must take for allocated elements.

allocator::allocate:

```
pointer allocate(size_type count,
    typename allocator<void>::const_pointer *hint = 0);
```

The member function allocates storage for an array of `count` elements of type `Ty`, by calling `operator new(count)`. It returns a pointer to the allocated object. The `hint` argument helps some allocators in improving locality of reference — a valid choice is the address of an object earlier allocated by the same allocator object, and not yet deallocated. To supply no hint, use a null pointer argument instead.

allocator::deallocate:

```
void deallocate(pointer ptr, size_type count);
```

The member function frees storage for the array of `count` objects of type `Ty` beginning at `ptr`, by calling `operator delete(ptr)`. The pointer `ptr` must have been earlier returned by a call to allocate for an allocator object that compares equal to `*this`, allocating an array object of the same size and type. `deallocate` never throws an exception.

allocator::destroy:

```
void destroy(pointer ptr);
```

The member function destroys the object designated by `ptr`, by calling the destructor `ptr->Ty::~Ty()`.

allocator::max_size:

```
size_type max_size() const throw();
```

The member function returns the length of the longest sequence of elements of type `Ty` that an object of class `allocator` *might* be able to allocate.

Operators:

allocator::operator=:

```
template<class Other>
    allocator& operator=(const allocator<Other>& right);
```

The template assignment operator does nothing. In general, however, an allocator object assigned to another allocator object should compare equal to it (and hence permit intermixing of object allocation and freeing between the two allocator objects).

allocator<void>

```
template<>
    class allocator<void> {
    typedef void *pointer;
    typedef const void *const_pointer;
    typedef void value_type;
    template<class Other>
        struct rebind;
    allocator() throw();
    template<class Other>
        allocator(const allocator<Other>) throw();
    template<class Other>
        allocator<void>& operator=(const allocator<Other>);
    };
```

The class explicitly specializes template class allocator for type *void.* Its constructors and assignment operator behave the same as for the template class, but it defines only the types const_pointer, pointer, value_type, and the nested template class rebind.

auto_ptr

Description: The class describes an object that stores a pointer to an allocated object **myptr** of type `Ty *`. The stored pointer must either be null or designate an object allocated by a `new` expression. An object constructed with a non-null pointer owns the pointer. It transfers ownership if its stored value is assigned to another object. (It replaces the stored value after a transfer with a null pointer.) The destructor for `auto_ptr<Ty>` deletes the allocated object if it owns it. Hence, an object of class `auto_ptr<Ty>` ensures that an allocated object is automatically deleted when control leaves a block, even via a thrown exception. You should not construct two `auto_ptr<Ty>` objects that own the same object.

You can pass an `auto_ptr<Ty>` object by value as an argument to a function call. You can return such an object by value as well. (Both operations depend on the implicit construction of intermediate objects of class `auto_ptr_ref<Ty>`, by various subtle conversion rules.) You cannot, however, reliably manage a sequence of `auto_ptr<Ty>` objects with an STL container.

Synopsis:

```
template<class Ty>
    class auto_ptr {
public:
    typedef Ty element_type;
    explicit auto_ptr(Ty *ptr = 0) throw();
    auto_ptr(auto_ptr<Ty>& right) throw();
    template<class Other>
        auto_ptr(auto_ptr<Other>& right) throw();
    auto_ptr(auto_ptr_ref<Ty> right) throw();
    ~auto_ptr();
    template<class Other>
        operator auto_ptr<Other>() throw();
    template<class Other>
        operator auto_ptr_ref<Other>() throw();
    template<class Other>
        auto_ptr<Ty>& operator=(auto_ptr<Other>& right) throw();
    auto_ptr<Ty>& operator=(auto_ptr<Ty>& right) throw();
```

```
    auto_ptr<Ty>& operator=(auto_ptr_ref<Ty>& right) throw();
    Ty& operator*() const throw();
    Ty *operator->() const throw();
    Ty *get() const throw();
    Ty *release() throw();
    void reset(Ty *ptr = 0);
    };
```

Constructor:

auto_ptr::auto_ptr:

```
explicit auto_ptr(Ty *ptr = 0) throw();
auto_ptr(auto_ptr<Ty>& right) throw();
auto_ptr(auto_ptr_ref<Ty> right) throw();
template<class Other>
    auto_ptr(auto_ptr<Other>& right) throw();
```

The first constructor stores `ptr` in `myptr`, the stored pointer to the allocated object. The second constructor transfers ownership of the pointer stored in `right`, by storing `right.release()`. in `myptr`. The third constructor behaves the same as the second, except that it stores `right.ref.release()` in `myptr`, where `ref` is the reference stored in `right`.

The template constructor behaves the same as the second constructor, provided that a pointer to `Other` can be implicitly converted to a pointer to `Ty`.

Destructor:

auto_ptr::~auto_ptr:

```
~auto_ptr();
```

The destructor evaluates the expression `delete myptr` to delete the object designated by the stored pointer.

Types:

auto_ptr::element_type:

```
typedef Ty element_type;
```

The type is a synonym for the template parameter `Ty`.

Classes:

auto_ptr_ref:

```
template<Class Ty>
    struct auto_ptr_ref {
    };
```

The class describes an object that stores a reference to an object of class `auto_ptr<Ty>` It is used as a helper class for `auto_ptr<Ty>` You should not have an occasion to construct an `auto_ptr_ref<Ty>` object directly.

Member functions:

auto_ptr::get:

```
Ty *get() const throw();
```

The member function returns the stored pointer `myptr`.

auto_ptr::reset:

```
void reset(Ty *ptr = 0);
```

The member function evaluates the expression `delete myptr`, but only if the stored pointer value `myptr` changes as a result of function call. It then replaces the stored pointer with `ptr`.

Operators:

auto_ptr::operator=:

```
template<class Other>
    auto_ptr<Ty>& operator=(auto_ptr<Other>& right) throw();
auto_ptr<Ty>& operator=(auto_ptr<>& right) throw();
auto_ptr<Ty>& operator=(auto_ptr_ref<>& right) throw();
```

The assignment evaluates the expression `delete myptr`, but only if the stored pointer `myptr` changes as a result of the assignment. It then transfers ownership of the pointer designated by `right`, by storing `right.release()` in `myptr`. (The last assignment behaves as if `right` designates the reference it stores.) The function returns `*this`.

auto_ptr::operator:*

```
Ty& operator*() const throw();
```

The indirection operator returns `*get()`. Hence, the stored pointer must not be null.

auto_ptr::operator->:

```
Ty *operator->() const throw();
```

The selection operator returns `get()`, so that the expression `ap->member` behaves the same as `(ap.get())->member`, where `ap` is an object of class `auto_ptr<Ty>`. Hence, the stored pointer must not be null, and `Ty` must be a class, structure, or union type with a member `member`.

auto_ptr::operator auto_ptr<Other>:

```
template<class Other>
    operator auto_ptr<Other>() throw();
```

The type cast operator returns `auto_ptr<Other>(*this)`.

auto_ptr::operator auto_ptr_ref<Other>:

```
template<class Other>
    operator auto_ptr_ref<Other>() throw();
```

The type cast operator returns `auto_ptr_ref<Other>(*this)`.

Members:

auto_ptr::release:

```
Ty *release() throw();
```

The member replaces the stored pointer `myptr` with a null pointer and returns the previously stored pointer.

bad_weak_ptr

```
class bad_weak_ptr
    : public std::exception {
public:
    bad_weak_ptr();
    const char *what() throw();
    };
```

[Added with TR1]

The class describes an exception that can be thrown from the "shared_ptr" on page 240 constructor that takes an argument of type "weak_ptr" on page 244. The member function what returns "tr1::bad_weak_ptr".

enable_shared_from_this

Description: [Added with TR1]

The template class can be used as a public base class to simplify creating "shared_ptr" on page 240 objects that own objects of the derived type:

```
class derived
    : public enable_shared_from_this<derived>
    {
    };

shared_ptr<derived> sp0 = new derived;
shared_ptr<derived> sp1 = sp0->shared_from_this();
```

The constructors, destructor, and assignment operator are protected to help prevent accidental misuse. The template argument type Ty must be the type of the derived class.

Synopsis:

```
template<class Ty>
    class enable_shared_from_this {
public:
    shared_ptr<Ty> shared_from_this();
    shared_ptr<const Ty> shared_from_this() const;

protected:
    enable_shared_from_this();
    enable_shared_from_this(const enable_shared_from_this&);
    enable_shared_from_this& operator=(const enable_shared_from_this&);
    ~enable_shared_from_this();
    };
```

Member functions:

enable_shared_from_this::shared_from_this:

```
shared_ptr<Ty> shared_from_this();
shared_ptr<const Ty> shared_from_this() const;
```

The member functions each return a "shared_ptr" on page 240 object that owns *(Ty*)this.

raw_storage_iterator

Description: The class describes an output iterator that constructs objects of type Ty in the sequence it generates. An object of class raw_storage_iterator<FwdIt, Ty> accesses storage through a forward iterator object, of class FwdIt, that you specify when you construct the object. For an object first of class FwdIt, the expression &*first must designate unconstructed storage for the next object (of type Ty) in the generated sequence.

Synopsis:

```
template<class FwdIt, class Ty>
    class raw_storage_iterator
        : public iterator<output_iterator_tag,
            void, void, void, void> {
public:
    explicit raw_storage_iterator(FwdIt first);
    raw_storage_iterator<FwdIt, Ty>& operator*();
    raw_storage_iterator<FwdIt, Ty>&
        operator=(const Ty& val);
    raw_storage_iterator<FwdIt, Ty>& operator++();
    raw_storage_iterator<FwdIt, Ty> operator++(int);
    };
```

Constructor:

raw_storage_iterator::raw_storage_iterator:

```
explicit raw_storage_iterator(FwdIt first);
```

The constructor stores first as the output iterator object.

Operators:

raw_storage_iterator::operator:*

```
raw_storage_iterator<FwdIt, Ty>& operator*();
```

The indirection operator returns *this (so that operator=(const Ty&) can perform the actual store in an expression such as *ptr = val).

raw_storage_iterator::operator=:

```
raw_storage_iterator<FwdIt, Ty>& operator=(const Ty& val);
```

The assignment operator constructs the next object in the output sequence using the stored iterator value first, by evaluating the placement new expression new ((void *)&*first) Ty(val). The function returns *this.

raw_storage_iterator::operator++:

```
raw_storage_iterator<FwdIt, Ty>& operator++();
raw_storage_iterator<FwdIt, Ty> operator++(int);
```

The first (preincrement) operator increments the stored output iterator object, then returns *this.

The second (postincrement) operator makes a copy of *this, increments the stored output iterator object, then returns the copy.

shared_ptr

Description: [Added with TR1]

The template class describes an object that uses reference counting to manage resources. Each shared_ptr object effectively holds a pointer to the resource that it owns or holds a null pointer. A resource can be owned by more than one shared_ptr object; when the last shared_ptr object that owns a particular resource is destroyed the resource is freed.

The template argument Ty may be an incomplete type except as noted for certain operand sequences.

When a shared_ptr<Ty> object is constructed from a resource pointer of type D* or from a shared_ptr<D>, the pointer type D* must be convertible to Ty*. If it is not, the code will not compile. For example:

```
class B {};
class D : public B {};

// okay, template parameter D and argument D*
shared_ptr<D> sp0(new D);

// okay, template parameter D and argument shared_ptr<D>
shared_ptr<D> sp1(sp0);

// okay, D* convertible to B*
shared_ptr<B> sp2(new D);

 // okay, template parameter B and argument shared_ptr<D>
shared_ptr<B> sp3(sp0);

// okay, template parameter B and argument shared_ptr<B>
shared_ptr<B> sp4(sp2);

// error, D* not convertible to int*
shared_ptr<int> sp4(new D);

// error, template parameter int and argument shared_ptr<B>
shared_ptr<int> sp5(sp2);
```

A shared_ptr object **owns** a resource:
- if it was constructed with a pointer to that resource,
- if it was constructed from a shared_ptr object that owns that resource,
- if it was constructed from a "weak_ptr" on page 244 object that points to that resource, or
- if ownership of that resource was assigned to it, either with operator= or by calling the member function reset.

All the shared_ptr objects that own a single resource share a control block which holds the number of shared_ptr objects that own the resource, the number of weak_ptr objects that point to the resource, and the deleter for that resource if it has one. A shared_ptr object that was initialized with a null pointer has a control block; thus it is not an empty shared_ptr. After a shared_ptr object **releases** a resource it no longer owns that resource. After a weak_ptr object releases a resource it no longer points to that resource. When the number of shared_ptr objects that own a resource becomes zero the resource is freed, either by deleting it or by passing its address to a deleter, depending on how ownership of the resource was originally created. When the number of shared_ptr objects that own a resource is zero and the number of weak_ptr objects that point to that resource is zero the control block is freed.

An **empty shared_ptr** object does not own any resources and has no control block.

A **deleter** is a function pointer or an object of a type with a member function `operator()`. Its type must be copy constructible and its copy constructor and destructor must not throw exceptions. A deleter is bound to a `shared_ptr` object with an operand sequence of the form `ptr, dtor`.

Some functions take an **operand sequence** that defines properties of the resulting `shared_ptr<Ty>` or `weak_ptr<Ty>` object. You can specify such an operand sequence several ways:

- *no arguments* — the resulting object is an empty shared_ptr object or an empty weak_ptr object.
- `ptr` — a pointer of type `Other*` to the resource to be managed. `Ty` must be a complete type. If the function fails it evaluates the expression `delete ptr`.
- `ptr, dtor` — a pointer of type `Other*` to the resource to be managed and a deleter for that resource. If the function fails it calls `dtor(ptr)`, which must be well defined.
- `sp` — a `shared_ptr<Other>` object that owns the resource to be managed.
- `wp` — a `weak_ptr<Other>` object that points to the resource to be managed.
- `ap` — an `auto_ptr<Other>` object that holds a pointer to the resource to be managed. If the function succeeds it calls `ap.release()`; otherwise it leaves `ap` unchanged.

In all cases, the pointer type `Other*` must be convertible to `Ty*`.

Synopsis:

```
template<class Ty>
    class shared_ptr {
public:
    typedef Ty element_type;

    shared_ptr();
    template<class Other>
        explicit shared_ptr(Other*);
    template<class Other, class D>
        shared_ptr(Other*, D);
    shared_ptr(const shared_ptr&);
    template<class Other>
        shared_ptr(const shared_ptr<Other>&);
    template<class Other>
        shared_ptr(const weak_ptr<Other>&);
    template<class &>
        shared_ptr(const std::auto_ptr<Other>&);
    ~shared_ptr();

    shared_ptr& operator=(const shared_ptr&);
    template<class Other>
        shared_ptr& operator=(const shared_ptr<Other>&);
    template<class Other>
        shared_ptr& operator=(auto_ptr<Other>&);

    void swap(shared_ptr&);
    void reset();
    template<class Other>
        void reset(Other*);
    template<class Other, class D>
        void reset(Other*, D);

    Ty *get() const;
    Ty& operator*() const;
    Ty *operator->() const;
```

```
    long use_count() const;
    bool unique() const;
    operator boolean-type() const;
    };
```

Constructor:

shared_ptr::shared_ptr:

```
shared_ptr();
template<class Other>
    explicit shared_ptr(Other *ptr);
template<class Other, class D>
    shared_ptr(Other *ptr, D dtor);
shared_ptr(const shared_ptr& sp);
template<class Other>
    shared_ptr(const shared_ptr<Other>& sp);
template<class Other>
    shared_ptr(const weak_ptr<Other>& wp);
template<class Other>
    shared_ptr(const std::auto_ptr<Other>& ap);
```

The constructors each construct an object that owns the resource named by the operand sequence. The constructor `shared_ptr(const weak_ptr<Other>& wp)` throws an exception object of type "bad_weak_ptr" on page 239 if `wp.expired()`.

Destructor:

shared_ptr::~shared_ptr:

```
~shared_ptr();
```

The destructor releases the resource owned by `*this`.

Types:

shared_ptr::element_type:

```
typedef Ty element_type;
```

The type is a synonym for the template parameter `Ty`.

Member functions:

shared_ptr::get:

```
Ty *get() const;
```

The member function returns the address of the owned resource. If the object does not own a resource it returns 0.

shared_ptr::reset:

```
void reset();
template<class Other>
    void reset(Other *ptr;);
template<class Other, class D>
    void reset(Other *ptr, D dtor);
```

The member functions all release the resource currently owned by `*this` and assign ownership of the resource named by the operand sequence to `*this`. If a member function fails it leaves `*this` unchanged.

shared_ptr::swap:

```
void swap(shared_ptr& sp);
```

The member function leaves the resource originally owned by `*this` subsequently owned by `sp`, and the resource originally owned by `sp` subsequently owned by `*this`. The function does not change the reference counts for the two resources and it does not throw any exceptions.

shared_ptr::unique:

```
bool unique() const;
```

The member function returns `true` if no other `shared_ptr` object owns the resource that is owned by `*this`, otherwise `false`.

shared_ptr::use_count:

```
long use_count() const;
```

The member function returns the number of `shared_ptr` objects that own the resource that is owned by `*this`.

Operators:

shared_ptr::operator=:

```
shared_ptr& operator=(const shared_ptr& sp);
template<class Other>
    shared_ptr& operator=(const shared_ptr<Other>& sp);
template<class Other>
    shared_ptr& operator=(auto_ptr<Other>& ap);
```

The operators all release the resource currently owned by `*this` and assign ownership of the resource named by the operand sequence to `*this`. If an operator fails it leaves `*this` unchanged.

shared_ptr::operator:*

```
Ty& operator*() const;
```

The indirection operator returns `*get()`. Hence, the stored pointer must not be null.

shared_ptr::operator->:

```
Ty *operator->() const;
```

The selection operator returns `get()`, so that the expression `sp->member` behaves the same as `(sp.get())->member` where `sp` is an object of class `shared_ptr<Ty>`. Hence, the stored pointer must not be null, and `Ty` must be a class, structure, or union type with a member `member`.

shared_ptr::operator boolean-type:

```
operator boolean-type() const;
```

The operator returns a value of a type that is convertible to `bool`. The result of the conversion to `bool` is `true` when `get() != 0`, otherwise `false`.

weak_ptr

Description: [Added with TR1]

The template class describes an object that points to a resource that is managed by one or more "shared_ptr" on page 240 objects. The weak_ptr objects that point to a resource do not affect the resource's reference count. Thus, when the last shared_ptr object that manages that resource is destroyed the resource will be freed, even if there are weak_ptr objects pointing to that resource. This is essential for avoiding cycles in data structures.

A weak_ptr object **points to** a resource if it was constructed from a shared_ptr object that owns that resource, if it was constructed from a weak_ptr object that points to that resource, or if that resource was assigned to it with operator=. A weak_ptr object does not provide direct access to the resource that it points to. Code that needs to use the resource does so through a shared_ptr object that owns that resource, created by calling the member function lock. A weak_ptr object has **expired** when the resource that it points to has been freed because all of the shared_ptr objects that own the resource have been destroyed. Calling lock on a weak_ptr object that has expired creates an empty shared_ptr object.

An **empty weak_ptr** object does not point to any resources and has no control block. Its member function lock returns an empty shared_ptr object.

A **cycle** occurs when two or more resources controlled by shared_ptr objects hold mutually referencing shared_ptr objects. For example, a circular linked list with three elements has a head node N0; that node holds a shared_ptr object that owns the next node, N1; that node holds a shared_ptr object that owns the next node, N2; that node, in turn, holds a shared_ptr object that owns the head node, N0, closing the cycle. In this situation, none of the reference counts will every become zero, and the nodes in the cycle will not be freed. To eliminate the cycle, the last node N2 should hold a weak_ptr object pointing to N0 instead of a smart_ptr object. Since the weak_ptr object does not own N0 it doesn't affect N0's reference count, and when the program's last reference to the head node is destroyed the nodes in the list will also be destroyed.

Synopsis:

```
template<class Ty> class weak_ptr {
public:
    typedef Ty element_type;

    weak_ptr();
    weak_ptr(const weak_ptr&);
    template<class Other>
        weak_ptr(const weak_ptr<Other>&);
    template<class Other>
        weak_ptr(const shared_ptr<Other>&);

    weak_ptr& operator=(const weak_ptr&);
    template<class Other>
        weak_ptr& operator=(const weak_ptr<Other>&);
    template<class Other>
        weak_ptr& operator=(shared_ptr<Other>&);

    void swap(weak_ptr&);
    void reset();

    long use_count() const;
    bool expired() const;
    shared_ptr<Ty> lock() const;
    };
```

Constructor:

weak_ptr::weak_ptr:

```
weak_ptr();
weak_ptr(const weak_ptr& wp);
template<class Other>
    weak_ptr(const weak_ptr<Other>& wp);
template<class Other>
    weak_ptr(const shared_ptr<Other>& sp);
```

The constructors each construct an object that points to the resource named by the operand sequence.

Types:

weak_ptr::element_type:

```
typedef Ty element_type;
```

The type is a synonym for the template parameter `Ty`.

Member functions:

weak_ptr::expired:

```
bool expired() const;
```

The member function returns `true` if `*this` has expired, otherwise `false`.

weak_ptr::lock:

```
shared_ptr<Ty> lock() const;
```

The member function returns an empty shared_ptr object if `*this` has expired; otherwise it returns a `"shared_ptr" on page 240<Ty>` object that owns the resource that `*this` points to.

weak_ptr::reset:

```
void reset();
```

The member function releases the resource pointed to by `*this` and converts `*this` to an empty weak_ptr object.

weak_ptr::swap:

```
void swap(weak_ptr& wp);
```

The member function leaves the resource originally pointed to by `*this` subsequently pointed to by `wp`, and the resource originally pointed to by `wp` subsequently pointed to by `*this`. The function does not change the reference counts for the two resources and it does not throw any exceptions.

weak_ptr::use_count:

```
long use_count() const;
```

The member function returns the number of `shared_ptr` objects that own the resource pointed to by `*this`.

Operators:

weak_ptr::operator=:

```
weak_ptr& operator=(const weak_ptr& wp);
template<class Other>
    weak_ptr& operator=(const weak_ptr<Other>& wp);
template<class Other>
    weak_ptr& operator=(const shared_ptr<Other>& sp);
```

The operators all release the resource currently pointed to by *this and assign ownership of the resource named by the operand sequence to *this. If an operator fails it leaves *this unchanged.

Functions

const_pointer_cast

```
template <class Ty, class Other>
    shared_ptr<Ty> const_pointer_cast(const shared_ptr<Other>& sp);
```

[Added with TR1]

The template function returns an empty shared_ptr object if const_cast<Ty*>(sp.get()) returns a null pointer; otherwise it returns a "shared_ptr" on page 240<Ty> object that owns the resource that is owned by sp. The expression const_cast<Ty*>(sp.get()) must be valid.

dynamic_pointer_cast

```
template <class Ty, class Other>
    shared_ptr<Ty> dynamic_pointer_cast(const shared_ptr<Other>& sp);
```

[Added with TR1]

The template function returns an empty shared_ptr object if dynamic_cast<Ty*>(sp.get()) returns a null pointer; otherwise it returns a "shared_ptr" on page 240<Ty> object that owns the resource that is owned by sp. The expression dynamic_cast<Ty*>(sp.get()) must be valid.

get_deleter

```
template<class D, class Ty>
    D *get_deleter(const shared_ptr<Ty>& sp);
```

[Added with TR1]

The template function returns a pointer to the deleter of type D that belongs to the "shared_ptr" on page 240 object sp. If sp has no deleter or if its deleter is not of type D the function returns 0.

get_temporary_buffer

```
template<class Ty>
    pair<Ty *, ptrdiff_t>
        get_temporary_buffer(ptrdiff_t count);
```

The template function allocates storage for a sequence of at most count elements of type Ty, from an unspecified source (which may well be the standard heap used by operator new). It returns a value pr, of type pair<Ty *, ptrdiff_t>. If the function allocates storage, pr.first designates the allocated storage and pr.second is the number of elements in the longest sequence the storage can hold. Otherwise, pr.first is a null pointer.

In this implementation, if a translator does not support member template functions, the template:

```
template<class Ty>
    pair<Ty *, ptrdiff_t>
        get_temporary_buffer(ptrdiff_t count);
```

is replaced by:

```
template<class Ty>
    pair<Ty *, ptrdiff_t>
        get_temporary_buffer(ptrdiff_t count, Ty *);
```

operator<

```
template<class Ty1, class Ty2>
    bool operator<(const shared_ptr<Ty1>& left, const shared_ptr<Ty2>& right);
template<class Ty1, class Ty2>
    bool operator<(const weak_ptr<Ty1>& left, const weak_ptr<Ty2>& right);
```

[Added with TR1]

The template functions impose a strict weak ordering on objects of their respective types. The actual ordering is implementation-specific, except that `!(left < right) && !(right < left)` is true only when `left` and `right` own or point to the same resource. The ordering imposed by these functions enables the use of "shared_ptr" on page 240 and "weak_ptr" on page 244 objects as keys in associative containers.

operator<<

```
template<class Elem, class Tr, class Ty>
    std::basic_ostream<Elem, Tr>& operator<<(std::basic_ostream<Elem, Tr>& out,
    shared_ptr<Ty>& sp);
```

[Added with TR1]

The template function returns `out << sp.get()`.

return_temporary_buffer

```
template<class Ty>
    void return_temporary_buffer(Ty *pbuf);
```

The template function frees the storage designated by `pbuf`, which must be earlier allocated by a call to get_temporary_buffer.

static_pointer_cast

```
template <class Ty, class Other>
    shared_ptr<Ty> static_pointer_cast(const shared_ptr<Other>& sp);
```

[Added with TR1]

The template function returns an empty shared_ptr object if `sp` is an empty `shared_ptr` object; otherwise it returns a "shared_ptr" on page 240<Ty> object that owns the resource that is owned by `sp`. The expression `static_cast<Ty*>(sp.get())` must be valid.

swap

```
template<class Ty, class Other>
    void swap(shared_ptr<Ty>& left, shared_ptr<Other>& right);
template<class Ty, class Other>
    void swap(weak_ptr<Ty>& left, weak_ptr<Other>& right);
```

[Added with TR1]

The template functions call `left.swap(right)`.

uninitialized_copy

```
template<class InIt, class FwdIt>
    FwdIt uninitialized_copy(InIt first, InIt last,
        FwdIt dest);
```

The template function effectively executes:

```
while (first != last)
    new ((void *)&*dest++)
        iterator_traits<InIt>::value_type(*first++);
return first;
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

uninitialized_fill

```
template<class FwdIt, class Ty>
    void uninitialized_fill(FwdIt first, FwdIt last,
        const Ty& val);
```

The template function effectively executes:

```
while (first != last)
    new ((void *)&*first++)
        iterator_traits<FwdIt>::value_type(val);
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

uninitialized_fill_n

```
template<class FwdIt, class Size, class Ty>
    void uninitialized_fill_n(FwdIt first, Size count,
        const Ty& val);
```

The template function effectively executes:

```
while (0 < count--)
    new ((void *)&*first++)
        iterator_traits<FwdIt>::value_type(val);
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

Operators

operator!=

```
template<class Ty>
    bool operator!=(const allocator<Ty>& left,
        const allocator<Ty>& right) throw();
template<class Ty1, class Ty2>
    bool operator!=(const shared_ptr<Ty1>& left,
        const shared_ptr<Ty2>& right);
```

The first template operator returns false. (All default allocators are equal.) The second template operator returns `!(left == right)`.

operator==

```
template<class Ty>
    bool operator==(const allocator<Ty>& left,
        const allocator<Ty>& right) throw();
template<class Ty1, class Ty2>
    bool operator==(const shared_ptr<Ty1>& left,
        const shared_ptr<Ty2>& right);
```

The first template operator returns true. (All default allocators are equal.) The second template operator returns `left.get() == right.get()`.

<new>

Description

Include the standard header **<new>** to define several types and functions that control allocation and freeing of storage under program control.

Some of the functions declared in this header are **replaceable**. The implementation supplies a default version, whose behavior is described in this document. A program can, however, define a function with the same signature to replace the default version at link time. The replacement version must satisfy the requirements described in this document.

Synopsis

```
namespace std {
typedef void (*new_handler)();
class bad_alloc;
class nothrow_t;
extern const nothrow_t nothrow;

        // FUNCTIONS
new_handler set_new_handler(new_handler ph) throw();
    }

        // OPERATORS -- NOT IN NAMESPACE std
void operator delete(void *p) throw();
void operator delete(void *, void *) throw();
void operator delete(void *p,
    const std::nothrow_t&) throw();
void operator delete[](void *p) throw();
void operator delete[](void *, void *) throw();
void operator delete[](void *p,
    const std::nothrow_t&) throw();
void *operator new(std::size_t n)
    throw(std::bad_alloc);
void *operator new(std::size_t n,
    const std::nothrow_t&) throw();
void *operator new(std::size_t n, void *p) throw();
void *operator new[](std::size_t n)
    throw(std::bad_alloc);
void *operator new[](std::size_t n,
    const std::nothrow_t&) throw();
void *operator new[](std::size_t n, void *p) throw();
```

Macros

__IBM_ALLOW_OVERRIDE_PLACEMENT_NEW

```
#define __IBM_ALLOW_OVERRIDE_PLACEMENT_NEW
```

The __IBM_ALLOW_OVERRIDE_PLACEMENT_NEW macro can be used to obtain the pre-z/OS V1R10 behavior of global placement new operators. By default, this macro is not predefined, and the Standard C++ Library provides the inlined version of global placement new operators, which can assist with improving performance.

Classes

bad_alloc

Description: The class describes an exception thrown to indicate that an allocation request did not succeed. The value returned by `what()` is an implementation-defined C string. None of the member functions throw any exceptions.

Synopsis:

```
class bad_alloc : public exception {
    };
```

nothrow_t

Description: The class is used as a function parameter to `operator new` to indicate that the function should return a null pointer to report an allocation failure, rather than throw an exception.

Synopsis:

```
class nothrow_t {};
```

Functions

operator delete

```
void operator delete(void *p) throw();
void operator delete(void *, void *) throw();
void operator delete(void *p,
    const std::nothrow_t&) throw();
```

The first function is called by a **`delete expression`** to render the value of `p` invalid. The program can define a function with this function signature that replaces the default version defined by the Standard C++ Library. The required behavior is to accept a value of `p` that is null or that was returned by an earlier call to `operator new(size_t)`.

The default behavior for a null value of `p` is to do nothing. Any other value of `p` must be a value returned earlier by a call as described above. The default behavior for such a non-null value of `p` is to reclaim storage allocated by the earlier call. It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to `operator new(size_t)`, or to any of `calloc(size_t)`, `malloc(size_t)`, or `realloc(void*, size_t)`.

The second function is called by a **placement delete expression** corresponding to a `new expression` of the form `new(std::size_t)`. It does nothing.

The third function is called by a placement `delete` expression corresponding to a `new` expression of the form `new(std::size_t, const std::nothrow_t&)`. It calls `delete(p)`.

operator delete[]

```
void operator delete[](void *p) throw();
void operator delete[](void *, void *) throw();
void operator delete[](void *p,
    const std::nothrow_t&) throw();
```

The first function is called by a **delete[] expression** to render the value of p invalid. The program can define a function with this function signature that replaces the default version defined by the Standard C++ Library.

The required behavior is to accept a value of p that is null or that was returned by an earlier call to operator new[](size_t).

The default behavior for a null value of p is to do nothing. Any other value of ptr must be a value returned earlier by a call as described above. The default behavior for such a non-null value of p is to reclaim storage allocated by the earlier call. It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to operator new(size_t), or to any of calloc(size_t), malloc(size_t), or realloc(void*, size_t).

The second function is called by a **placement delete[] expression** corresponding to a new[] expression of the form new[](std::size_t). It does nothing.

The third function is called by a placement delete expression corresponding to a new[] expression of the form new[](std::size_t, const std::nothrow_t&). It calls delete[](p).

operator new

```
void *operator new(std::size_t n) throw(bad_alloc);
void *operator new(std::size_t n,
    const std::nothrow_t&) throw();
void *operator new(std::size_t n, void *p) throw();
```

The first function is called by a **new expression** to allocate n bytes of storage suitably aligned to represent any object of that size. The program can define a function with this function signature that replaces the default version defined by the Standard C++ Library.

The required behavior is to return a non-null pointer only if storage can be allocated as requested. Each such allocation yields a pointer to storage disjoint from any other allocated storage. The order and contiguity of storage allocated by successive calls is unspecified. The initial stored value is unspecified. The returned pointer points to the start (lowest byte address) of the allocated storage. If n is zero, the value returned does not compare equal to any other value returned by the function.

The default behavior is to execute a loop. Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to malloc(size_t) is unspecified. If the attempt is successful, the function returns a pointer to the allocated storage. Otherwise, the function calls the designated new handler. If the called function returns, the loop repeats. The loop terminates when an attempt to allocate the requested storage is successful or when a called function does not return.

The required behavior of a **new handler** is to perform one of the following operations:

- make more storage available for allocation and then return

- call either abort() or exit(int)
- throw an object of type bad_alloc

The default behavior of a new handler is to throw an object of type bad_alloc. A null pointer designates the default new handler.

The order and contiguity of storage allocated by successive calls to operator new(size_t) is unspecified, as are the initial values stored there.

The second function:

```
void *operator new(std::size_t n,
    const std::nothrow_t&) throw();
```

is called by a placement new expression to allocate n bytes of storage suitably aligned to represent any object of that size. The program can define a function with this function signature that replaces the default version defined by the Standard C++ Library.

The default behavior is to return operator new(n) if that function succeeds. Otherwise, it returns a null pointer.

The third function:

```
void *operator new(std::size_t n, void *p) throw();
```

is called by a **placement new expression**, of the form new (args) T. Here, args consists of a single object pointer. The function returns p.

operator new[]

```
void *operator new[](std::size_t n)
    throw(std::bad_alloc);
void *operator new[](std::size_t n,
    const std::nothrow_t&) throw();
void *operator new[](std::size_t n, void *p) throw();
```

The first function is called by a **new[] expression** to allocate n bytes of storage suitably aligned to represent any array object of that size or smaller. The program can define a function with this function signature that replaces the default version defined by the Standard C++ Library.

The required behavior is the same as for operator new(size_t). The default behavior is to return operator new(n).

The second function is called by a placement new[] expression to allocate n bytes of storage suitably aligned to represent any array object of that size. The program can define a function with this function signature that replaces the default version defined by the Standard C++ Library.

The default behavior is to return operator new(n) if that function succeeds. Otherwise, it returns a null pointer.

The third function is called by a **placement new[] expression**, of the form new (args) T[N]. Here, args consists of a single object pointer. The function returns p.

set_new_handler

```
new_handler set_new_handler(new_handler ph) throw();
```

The function stores ph in a static new handler pointer that it maintains, then returns the value previously stored in the pointer. The new handler is used by operator new(size_t).

Types

new_handler

```
typedef void (*new_handler)();
```

The type points to a function suitable for use as a new handler.

Objects

nothrow

```
extern const nothrow_t nothrow;
```

The object is used as a function argument to match the parameter type nothrow_t.

<numeric>

Description

Include the STL standard header **<numeric>** to define several template functions useful for computing numeric values. The descriptions of these templates employ a number of conventions common to all algorithms.

Synopsis

```
namespace std {
template<class InIt, class T>
    T accumulate(InIt first, InIt last, T val);
template<class InIt, class T, class Pred>
    T accumulate(InIt first, InIt last, T val, Pred pr);
template<class InIt1, class InIt2, class T>
    T inner_product(InIt1 first1, InIt1 last1,
        Init2 first2, T val);
template<class InIt1, class InIt2, class T,
    class Pred1, class Pred2>
    T inner_product(InIt1 first1, InIt1 last1,
        Init2 first2, T val, Pred1 pr1, Pred2 pr2);
template<class InIt, class OutIt>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result, Pred pr);
template<class InIt, class OutIt>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result, Pred pr);
    }
```

Template functions

accumulate

```
template<class InIt, class T>
    T accumulate(InIt first, InIt last, T val);
template<class InIt, class T, class Pred>
    T accumulate(InIt first, InIt last, T val, Pred pr);
```

The first template function repeatedly replaces `val` with `val + *I`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. It then returns `val`.

The second template function repeatedly replaces `val` with `pr(val, *I)`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. It then returns `val`.

adjacent_difference

```
template<class InIt, class OutIt>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result, Pred pr);
```

The first template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value stored is `*I - val`, and `val` is replaced by `*I`. The function returns `result` incremented `last - first` times.

The second template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value stored is `pr(*I, val)`, and `val` is replaced by `*I`. The function returns `result` incremented `last - first` times.

inner_product

```
template<class InIt1, class InIt2, class T>
    T inner_product(InIt1 first1, InIt1 last1,
        Init2 first2, T val);
template<class InIt1, class InIt2, class T,
    class Pred1, class Pred2>
    T inner_product(InIt1 first1, InIt1 last1,
        Init2 first2, T val, Pred1 pr1, Pred2 pr2);
```

The first template function repeatedly replaces `val` with `val + (*I1 * *I2)`, for each value of the `InIt1` iterator `I1` in the interval `[first1, last2)`. In each case, the `InIt2` iterator `I2` equals `first2 + (I1 - first1)`. The function returns `val`.

The second template function repeatedly replaces `val` with `pr1(val, pr2(*I1, *I2))`, for each value of the `InIt1` iterator `I1` in the interval `[first1, last2)`. In each case, the `InIt2` iterator `I2` equals `first2 + (I1 - first1)`. The function returns `val`.

partial_sum

```
template<class InIt, class OutIt>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result, Pred pr);
```

The first template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value `val` stored is `val + *I`. The function returns `result` incremented `last - first` times.

The second template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value `val` stored is `pr(val, *I)`. The function returns `result` incremented `last - first` times.

<ostream>

Description

Include the iostreams standard header **<ostream>** to define template class basic_ostream, which mediates insertions for the iostreams. The header also defines several related manipulators. (This header is typically included for you by another of the iostreams headers. You seldom have occasion to include it directly.)

Synopsis

```
namespace std {
template<class E, class T = char_traits<E> >
    class basic_ostream;
typedef basic_ostream<char, char_traits<char> >
    ostream;
typedef basic_ostream<wchar_t, char_traits<wchar_t> >
    wostream;

        // INSERTERS
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const E *s);
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            E c);
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const char *s);
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            char c);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const char *s);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            char c);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const signed char *s);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            signed char c);
template<class T>
```

```
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const unsigned char *s);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            unsigned char c);

        // MANIPULATORS
template class<E, T>
    basic_ostream<E, T>&
        endl(basic_ostream<E, T>& os);
template class<E, T>
    basic_ostream<E, T>&
        ends(basic_ostream<E, T>& os);
template class<E, T>
    basic_ostream<E, T>&
        flush(basic_ostream<E, T>& os);
    }
```

Classes

basic_ostream

Description: The template class describes an object that controls insertion of elements and encoded objects into a stream buffer with elements of type E, also known as char_type, whose character traits are determined by the class T, also known as traits_type.

Most of the member functions that overload operator<< are **formatted output functions**. They follow the pattern:

```
    iostate state = goodbit;
    const sentry ok(*this);
    if (ok)
        {try
            {<convert and insert elements
            accumulate flags in state>}
        catch (...)
            {try
                {setstate(badbit); }
   catch (...)
    {}
            if ((exceptions() & badbit) != 0)
                throw; }}
    width(0);    // except for operator<<(E)
    setstate(state);
    return (*this);
```

Two other member functions are **unformatted output functions**. They follow the pattern:

```
    iostate state = goodbit;
    const sentry ok(*this);
    if (!ok)
        state |= badbit;
    else
        {try
            {<obtain and insert elements
            accumulate flags in state>}
        catch (...)
            {try
                {setstate(badbit); }
   catch (...)
    {}
```

```
        if ((exceptions() & badbit) != 0)
            throw; }}
    setstate(state);
    return (*this);
```

Both groups of functions call `setstate(badbit)` if they encounter a failure while inserting elements.

An object of class `basic_istream<E, T>` stores only a virtual public base object of class **basic_ios<E, T>**

Synopsis:

```
template <class E, class T = char_traits<E> >
    class basic_ostream
        : virtual public basic_ios<E, T> {
public:
    typedef typename basic_ios<E, T>::char_type char_type;
    typedef typename basic_ios<E, T>::traits_type traits_type;
    typedef typename basic_ios<E, T>::int_type int_type;
    typedef typename basic_ios<E, T>::pos_type pos_type;
    typedef typename basic_ios<E, T>::off_type off_type;
    explicit basic_ostream(basic_streambuf<E, T> *sb);
    class sentry;
    virtual ~ostream();
    bool opfx();
    void osfx();
    basic_ostream& operator<<(
        basic_ostream& (*pf)(basic_ostream&));
    basic_ostream& operator<<(
        ios_base;& (*pf)(ios_base&));
    basic_ostream& operator<<(
        basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
    basic_ostream& operator<<(
        basic_streambuf<E, T> *sb);
    basic_ostream& operator<<(bool n);
    basic_ostream& operator<<(short n);
    basic_ostream& operator<<(unsigned short n);
    basic_ostream& operator<<(int n);
    basic_ostream& operator<<(unsigned int n);
    basic_ostream& operator<<(long n);
    basic_ostream& operator<<(unsigned long n);
    basic_ostream& operator<<(float n);
    basic_ostream& operator<<(double n);
    basic_ostream& operator<<(long double n);
    basic_ostream& operator<<(const void *n);
    basic_ostream& put(char_type c);
    basic_ostream& write(char_type *s, streamsize n);
    basic_ostream& flush();
    pos_type tellp();
    basic_ostream& seekp(pos_type pos);
    basic_ostream& seekp(off_type off,
        ios_base::seek_dir way);
    };
```

Constructor:

basic_ostream::basic_ostream:

```
explicit basic_ostream(basic_streambuf<E, T> *sb);
```

The constructor initializes the base class by calling `init(sb)`.

Member classes:

basic_ostream::sentry:

```
class sentry {
public:
    explicit sentry(basic_ostream<E, T>& os);
    operator bool() const;
private:
    sentry(const sentry&);  // not defined
    sentry& operator=(const sentry&);  // not defined
    };
```

The nested class describes an object whose declaration structures the formatted output functions and the unformatted output functions. The constructor effectively calls `os.opfx()` and stores the return value. `operator bool()` delivers this return value. The destructor effectively calls `os.osfx()`, but only if `uncaught_exception()` returns false.

Member functions:

basic_ostream::flush:

```
basic_ostream& flush();
```

If `rdbuf()` is not a null pointer, the function calls `rdbuf()->pubsync()`. If that returns -1, the function calls `setstate(badbit)`. It returns `*this`.

basic_ostream::operator<<:

```
basic_ostream& operator<<(
    basic_ostream& (*pf)(basic_ostream&));
basic_ostream& operator<<(
    ios_base& (*pf)(ios_base&));
basic_ostream& operator<<(
    basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
basic_ostream& operator<<(
    basic_streambuf<E, T> *sb);
basic_ostream& operator<<(bool n);
basic_ostream& operator<<(short n);
basic_ostream& operator<<(unsigned short n);
basic_ostream& operator<<(int n);
basic_ostream& operator<<(unsigned int n);
basic_ostream& operator<<(long n);
basic_ostream& operator<<(unsigned long n);
basic_ostream& operator<<(float n);
basic_ostream& operator<<(double n);
basic_ostream& operator<<(long double n);
basic_ostream& operator<<(const void *n);
```

The first member function ensures that an expression of the form `ostr << endl` calls `endl(ostr)`, then returns `*this`. The second and third functions ensure that other manipulators, such as hex behave similarly. The remaining functions are all formatted output functions.

The function:

```
basic_ostream& operator<<(
    basic_streambuf<E, T> *sb);
```

extracts elements from `sb`, if `sb` is not a null pointer, and inserts them. Extraction stops on end-of-file, or if an extraction throws an exception (which is rethrown). It also stops, without extracting the element in question, if an insertion fails. If the function inserts no elements, or if an extraction throws an exception, the function calls `setstate(failbit)`. In any case, the function returns `*this`.

The function:

```
basic_ostream& operator<<(bool n);
```

converts `n` to a boolean field and inserts it by calling `use_facet<num_put<E, OutIt>(getloc()). put(OutIt( rdbuf()), *this, getloc(), n)`. Here, `OutIt` is defined as `ostreambuf_iterator<E, T>`. The function returns `*this`.

The functions:

```
basic_ostream& operator<<(short n);
basic_ostream& operator<<(unsigned short n);
basic_ostream& operator<<(int n);
basic_ostream& operator<<(unsigned int n);
basic_ostream& operator<<(long n);
basic_ostream& operator<<(unsigned long n);
basic_ostream& operator<<(const void *n);
```

each convert `n` to a numeric field and insert it by calling `use_facet<num_put<E, OutIt>(getloc()). put(OutIt( rdbuf()), *this, getloc(), n)`. Here, `OutIt` is defined as `ostreambuf_iterator<E, T>`.

The function returns `*this`.

The functions:

```
basic_ostream& operator<<(float n);
basic_ostream& operator<<(double n);
basic_ostream& operator<<(long double n);
```

each convert `n` to a numeric field and insert it by calling `use_facet<num_put<E, OutIt>(getloc()). put(OutIt( rdbuf()), *this, getloc(), n)`. Here, `OutIt` is defined as `ostreambuf_iterator<E, T>`. The function returns `*this`.

basic_ostream::opfx:

```
bool opfx();
```

If `good()` is true, and `tie()` is not a null pointer, the member function calls `tie->flush()`. It returns `good()`.

You should not call `opfx` directly. It is called as needed by an object of class sentry.

basic_ostream::osfx:

```
void osfx();
```

If `flags() & unitbuf` is nonzero, the member function calls `flush()`. You should not call `osfx` directly. It is called as needed by an object of class `sentry`.

basic_ostream::put:

```
basic_ostream& put(char_type c);
```

The unformatted output function inserts the element `c`. It returns `*this`.

basic_ostream::seekp:

```
basic_ostream& seekp(pos_type pos);
basic_ostream& seekp(off_type off,
    ios_base::seek_dir way);
```

If fail() is false, the first member function calls rdbuf()-> pubseekpos(pos). If fail() is false, the second function calls rdbuf()-> pubseekoff(off, way). Both functions return *this.

basic_ostream::tellp:

```
pos_type tellp();
```

If fail() is false, the member function returns rdbuf()-> pubseekoff(0, cur, in). Otherwise, it returns pos_type(-1).

basic_ostream::write:

```
basic_ostream& write(const char_type *s, streamsize n);
```

The unformatted output function inserts the sequence of n elements beginning at s.

Template functions

operator<<

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const E *s);
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            E c);
    template<class E, class T>
        basic_ostream<E, T>&
            operator<<(basic_ostream<E, T>& os,
                const char *s);
    template<class E, class T>
        basic_ostream<E, T>&
            operator<<(basic_ostream<E, T>& os,
                char c);
    template<class T>
        basic_ostream<char, T>&
            operator<<(basic_ostream<char, T>& os,
                const char *s);
    template<class T>
        basic_ostream<char, T>&
            operator<<(basic_ostream<char, T>& os,
                char c);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const signed char *s);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            signed char c);
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const unsigned char *s);
template<class T>
   basic_ostream<char, T>&
       operator<<(basic_ostream<char, T>& os,
           unsigned char c);
```

The template function:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const E *s);
```

is a formatted output functions that determines the length `n = traits_type::length(s)` of the sequence beginning at `s`, and inserts the sequence. If `n < os.width()`, then the function also inserts a repetition of `os.width() - n` fill characters. The repetition precedes the sequence if `(os.flags() & adjustfield != left`. Otherwise, the repetition follows the sequence. The function returns `os`.

The template function:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            E c);
```

inserts the element `c`. If `1 < os.width()`, then the function also inserts a repetition of `os.width() - 1` fill characters. The repetition precedes the sequence if `(os.flags() & adjustfield != left`. Otherwise, the repetition follows the sequence. It returns `os`.

The template function:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const char *s);
```

behaves the same as:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const E *s);
```

except that each element `c` of the sequence beginning at `s` is converted to an object of type `E` by calling `os.put(os. widen(c))`.

The template function:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            char c);
```

behaves the same as:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            E c);
```

except that `c` is converted to an object of type `E` by calling `os.put(os. widen(c))`.

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const char *s);
```

behaves the same as:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            const E *s);
```

(It does not have to widen the elements before inserting them.)

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            char c);
```

behaves the same as:

```
template<class E, class T>
    basic_ostream<E, T>&
        operator<<(basic_ostream<E, T>& os,
            E c);
```

(It does not have to widen `c` before inserting it.)

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const signed char *s);
```

returns `os << (const char *)s`.

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            signed char c);
```

returns `os << (char)c`.

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            const unsigned char *s);
```

returns `os << (const char *)s`.

The template function:

```
template<class T>
    basic_ostream<char, T>&
        operator<<(basic_ostream<char, T>& os,
            unsigned char c);
```

returns `os << (char)c`.

Manipulators

endl

```
template class<E, T>
    basic_ostream<E, T>& endl(basic_ostream<E, T>& os);
```

The manipulator calls os.put(os. widen('\n')), then calls os.flush(). It returns os.

ends

```
template class<E, T>
    basic_ostream<E, T>& ends(basic_ostream<E, T>& os);
```

The manipulator calls os.put(E('\0')). It returns os.

flush

```
template class<E, T>
    basic_ostream<E, T>& flush(basic_ostream<E, T>& os);
```

The manipulator calls os.flush(). It returns os.

Types

ostream

```
typedef basic_ostream<char, char_traits<char> > ostream;
```

The type is a synonym for template class basic_ostream, specialized for elements of type *char* with default character traits.

wostream

```
typedef basic_ostream<wchar_t, char_traits<wchar_t> >
    wostream;
```

The type is a synonym for template class basic_ostream, specialized for elements of type wchar_t with default character traits.

<queue>

Description

Include the STL standard header **<queue>** to define the template classes priority_queue and queue, and several supporting templates.

Synopsis

```
namespace std {
template<class T, class Cont>
    class queue;
template<class T, class Cont, class Pred>
    class priority_queue;

        // TEMPLATE FUNCTIONS
template<class T, class Cont>
    bool operator==(const queue<T, Cont>& lhs,
        const queue<T, Cont>&);
template<class T, class Cont>
    bool operator!=(const queue<T, Cont>& lhs,
        const queue<T, Cont>&);
template<class T, class Cont>
    bool operator<(const queue<T, Cont>& lhs,
        const queue<T, Cont>&);
template<class T, class Cont>
    bool operator>(const queue<T, Cont>& lhs,
        const queue<T, Cont>&);
template<class T, class Cont>
    bool operator<=(const queue<T, Cont>& lhs,
```

```
        const queue<T, Cont>&);
template<class T, class Cont>
    bool operator>=(const queue<T, Cont>& lhs,
        const queue<T, Cont>&);
    }
```

Classes

priority_queue

Description: The template class describes an object that controls a varying-length sequence of elements. The object allocates and frees storage for the sequence it controls through a protected object named **c**, of class `Cont`. The type `T` of elements in the controlled sequence must match value_type.

The sequence is ordered using a protected object named **comp**. After each insertion or removal of the top element (at position zero), for the iterators `P0` and `Pi` designating elements at positions `0` and `i`, `comp(*P0, *Pi)` is false. (For the default template parameter `less<typename Cont::value_type>` the top element of the sequence compares largest, or highest priority.)

An object of class `Cont` must supply random-access iterators and several public members defined the same as for deque and vector (both of which are suitable candidates for class `Cont`). The required members are:

```
    typedef T value_type;
    typedef T0 size_type;
    typedef T1 iterator;
    Cont();
    template<class InIt>
        Cont(InIt first, InIt last);
    template<class InIt>
        void insert(iterator it, InIt first, InIt last);
    iterator begin();
    iterator end();
    bool empty() const;
    size_type size() const;
    const value_type& front() const;
    void push_back(const value_type& x);
    void pop_back();
```

Here, `T0` and `T1` are unspecified types that meet the stated requirements.

Synopsis:

```
template<class T,
    class Cont = vector<T>,
    class Pred = less<typename Cont::value_type> >
    class priority_queue {
public:
    typedef Cont container_type;
    typedef typename Cont::value_type value_type;
    typedef typename Cont::size_type size_type;
    priority_queue();
    explicit priority_queue(const Pred& pr);
    priority_queue(const Pred& pr,
        const container_type& cont);
    priority_queue(const priority_queue& x);
    template<class InIt>
        priority_queue(InIt first, InIt last);
    template<class InIt>
        priority_queue(InIt first, InIt last,
            const Pred& pr);
    template<class InIt>
```

```
        priority_queue(InIt first, InIt last,
            const Pred& pr, const container_type& cont);
    bool empty() const;
    size_type size() const;
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
protected:
    Cont c;
    Pred comp;
    };
```

Constructor:

priority_queue::priority_queue:

```
priority_queue();
explicit priority_queue(const Pred& pr);
priority_queue(const Pred& pr,
    const container_type& cont);
priority_queue(const priority_queue& x);
template<class InIt>
    priority_queue(InIt first, InIt last);
template<class InIt>
    priority_queue(InIt first, InIt last,
        const Pred& pr);
template<class InIt>
    priority_queue(InIt first, InIt last,
        const Pred& pr, const container_type& cont);
```

All constructors with an argument `cont` initialize the stored object with `c(cont)`. The remaining constructors initialize the stored object with `c`, to specify an empty initial controlled sequence. The last three constructors then call `c.insert(c.end(), first, last)`.

All constructors also store a function object in comp. The function object `pr` is the argument `pr`, if present. For the copy constructor, it is `x.comp`. Otherwise, it is `Pred()`.

A non-empty initial controlled sequence is then ordered by calling `make_heap(c.begin(), c.end(), comp)`.

Types:

priority_queue::container_type:

```
typedef typename Cont::container_type container_type;
```

The type is a synonym for the template parameter `Cont`.

priority_queue::size_type:

```
typedef typename Cont::size_type size_type;
```

The type is a synonym for `Cont::size_type`.

priority_queue::value_type:

```
typedef typename Cont::value_type value_type;
```

The type is a synonym for `Cont::value_type`.

Member functions:

priority_queue::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

priority_queue::pop:

```
void pop();
```

The member function removes the first element of the controlled sequence, which must be non-empty, then reorders it.

priority_queue::push:

```
void push(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence, then reorders it.

priority_queue::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

priority_queue::top:

```
const value_type& top() const;
```

The member function returns a reference to the first (highest priority) element of the controlled sequence, which must be non-empty.

queue

Description: The template class describes an object that controls a varying-length sequence of elements. The object allocates and frees storage for the sequence it controls through a protected object named **c**, of class `Cont`. The type `T` of elements in the controlled sequence must match value_type.

An object of class `Cont` must supply several public members defined the same as for deque and list (both of which are suitable candidates for class `Cont`). The required members are:

```
typedef T value_type;
typedef T0 size_type;
Cont();
bool empty() const;
size_type size() const;
value_type& front();
const value_type& front() const;
value_type& back();
const value_type& back() const;
void push_back(const value_type& x);
void pop_front();
bool operator==(const Cont& X) const;
bool operator!=(const Cont& X) const;
bool operator<(const Cont& X) const;
bool operator>(const Cont& X) const;
bool operator<=(const Cont& X) const;
bool operator>=(const Cont& X) const;
```

Here, `T0` is an unspecified type that meets the stated requirements.

Synopsis:

```
template<class T,
    class Cont = deque<T> >
    class queue {
public:
    typedef Cont container_type;
    typedef typename Cont::value_type value_type;
    typedef typename Cont::size_type size_type;
    queue();
    explicit queue(const container_type& cont);
    bool empty() const;
    size_type size() const;
    value_type& back();
    const value_type& back() const;
    value_type& front();
    const value_type& front() const;
    void push(const value_type& x);
    void pop();
protected:
    Cont c;
    };
```

Constructor:

queue::queue:

```
queue();
explicit queue(const container_type& cont);
```

The first constructor initializes the stored object with `c()`, to specify an empty initial controlled sequence. The second constructor initializes the stored object with `c(cont)`, to specify an initial controlled sequence that is a copy of the sequence controlled by `cont`.

Types:

queue::container_type:

```
typedef Cont container_type;
```

The type is a synonym for the template parameter `Cont`.

queue::size_type:

```
typedef typename Cont::size_type size_type;
```

The type is a synonym for `Cont::size_type`.

queue::value_type:

```
typedef typename Cont::value_type value_type;
```

The type is a synonym for `Cont::value_type`.

Member functions:

queue::back:

```
value_type& back();
const value_type& back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

queue::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

queue::front:

```
value_type& front();
const value_type& front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

queue::pop:

```
void pop();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

queue::push:

```
void push(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence.

queue::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

Template functions

operator!=

```
template<class T, class Cont>
    bool operator!=(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class Cont>
    bool operator==(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function overloads `operator==` to compare two objects of template class queue. The function returns `lhs.c == rhs.c`.

operator<

```
template<class T, class Cont>
    bool operator<(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function overloads `operator<` to compare two objects of template class queue. The function returns `lhs.c < rhs.c`.

operator<=

```
template<class T, class Cont>
    bool operator<=(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class Cont>
    bool operator>(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class Cont>
    bool operator>=(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function returns `!(lhs < rhs)`.

<random>

Description

Include the TR1 header **<random>** to define a host of random number generators.

Note: To enable this header file, you must define the macro __IBMCPP_TR1__ and use the **TARGET** compiler option to specify a valid release level. Valid release levels for TR1 support are zOSV1R12 or later. For example, you can specify the **TARGET** option as follows: `TARGET(zOSV1R12)`.

Any release prior to zOSV1R12 is invalid for the use of TR1. If TR1 code is used in an application to be compiled on an earlier platform, the compiler will issue the `#error` directive.

Synopsis

```
namespace std {
    namespace tr1 {
    // UTILITIES
 template<class Engine,
    class Dist>
    class variate_generator;

    // SIMPLE ENGINES
 template<class UIntType,
    UIntType A, UIntType C, UIntType M>
    class linear_congruential;
 template<class UIntType,
    int W, int N, int M, int R,
    UIntType A, int U, int S,
    UIntType B, int T, UIntType C, int L>
    class mersenne_twister;
 template<class IntType,
    IntType M, int S, int R>
    class subtract_with_carry;
 template<class RealType,
    int W, int S, int R>
    class subtract_with_carry_01;
 class random_device;
```

```
    // COMPOUND ENGINES
 template<class Engine,
    int P, int R>
    class discard_block;
 template<class Engine1, int S1,
    class Engine2, int S2>
    class xor_combine;

    // ENGINES WITH PREDEFINED PARAMETERS
 typedef linear_congruential<unsigned long, 16807, 0, 2147483647> minstd_rand0;
 typedef linear_congruential<unsigned long, 48271, 0, 2147483647> minstd_rand;

 typedef mersenne_twister<ui-type, 32, 624,
    397, 31,0x9908b0df, 11, 7, 0x9d2c5680, 15, 0xefc60000, 18> mt19937;

 typedef subtract_with_carry_01<float, 24, 10, 24> ranlux_base_01;
 typedef subtract_with_carry_01<double, 48, 10, 24> ranlux64_base_01;

 typedef discard_block<subtract_with_carry<i-type,
    1 << 24, 10, 24>, 223, 24> ranlux3;

 typedef discard_block<subtract_with_carry<i-type,
    1 << 24, 10, 24>, 389, 24> ranlux4;

 typedef discard_block<subtract_with_carry_01<float, 24, 10, 24>,
    223, 24> ranlux3_01;

 typedef discard_block<subtract_with_carry_01<float, 24, 10, 24>,
    389, 24> ranlux4_01;

    // DISTRIBUTIONS
 template<class IntType = int>
    class uniform_int;
 template<class RealType = double>
    class bernoulli_distribution;
 template<class IntType = int, class RealType = double>
    class geometric_distribution;
 template<class IntType = int, class RealType = double>
    class poisson_distribution;
 template<class IntType = int, class RealType = double>
    class binomial_distribution;

 template<class RealType = double>
    class uniform_real;
 template<class RealType = double>
    class exponential_distribution;
 template<class RealType = double>
    class normal_distribution;
 template<class RealType = double>
    class gamma_distribution;
    }  // namespace tr1
}  // namespace std
```

Classes

bernoulli_distribution

Description: The class describes a distribution that produces values of type `bool`, returning `true` with a probability given by the argument to the constructor.

Synopsis:

```
class bernoulli_distribution {
public:
    typedef int input_type;
```

```
    typedef bool result_type;
    explicit bernoulli_distribution(double p0 = 0.5);
    double p() const;
    void reset();
    template<class Engine>
        result_type operator()(Engine& eng);
private:
    const double stored_p;  // exposition only
    };
```

Constructor:

bernoulli_distribution::bernoulli_distribution:

```
bernoulli_distribution(double p0 = 0.5);
```

Precondition: `0.0 ≤ p0 && p0 ≤ 1.0`

The constructor constructs an object whose stored value `stored_p` holds the value `p0`.

Member functions:

bernoulli_distribution::operator():

```
template<class Engine>
result_type operator()(Engine& eng);
```

The member function uses the engine `eng` as a source of uniformly distributed random integer values and returns `true` with probability given by the stored value `stored_p`.

bernoulli_distribution::p:

```
double p();
```

The member function returns the stored value `stored_p`.

binomial_distribution

Description: The template class describes a distribution that produces values of a user-specified integral type distributed with a binomial distribution.

Synopsis:

```
template<class IntType = int, class RealType = double>
    class binomial_distribution {
public:
    typedef /* implementation defined */ input_type;
    typedef IntType result_type;

    explicit binomial_distribution(result_type t0 = 1,
        const RealType& p0 = RealType(0.5));

    result_type t() const;
    RealType p() const;
    void reset();
    template<class Engine>
        result_type operator()(Engine& eng);
private:
    const result_type stored_t;  // exposition only
    const RealType stored_p;     // exposition only
    };
```

Constructor:

binomial_distribution::binomial_distribution:

```
binomial_distribution(result_type t0 = 1, const RealType& p0 = RealType(0.5));
```

Precondition: `0.0 <= t0 && 0.0 <= p0 && p0 <= 1.0`

The constructor constructs an object whose stored value `stored_p` holds the value `p0` and whose stored value `stored_t` holds the value `t0`.

Member functions:

binomial_distribution::operator():

```
template<class Engine>
result_type operator()(Engine& eng);
```

The member function uses the engine `eng` as a source of uniformly distributed random integral values and returns integral values with each value `i` occuring with probability `pr(i) = comb(stored_t, i) * stored_p`^{i} `* (1 - stored_p)`$^{stored\_t-i}$, where `comb(t, i)` is the number of possible combinations of `t` objects taken `i` at a time.

binomial_distribution::p:

```
RealType p();
```

The member function returns the stored value `stored_p`.

binomial_distribution::t:

```
result_type t();
```

The member function returns the stored value `stored_t`.

discard_block

Description: The template class describes a compound engine that produces values by discarding some of the values returned by its base engine. Each cycle of the compound engine begins by returning `R` values successively produced by the base engine and ends by discarding `P - R` such values. The engine's state is the state of `stored_eng` followed by the number of calls to `operator()` that have occurred since the beginning of the current cycle.

The value of the template argument `R` must be less than or equal to the value of the template argument `P`.

Synopsis:

```
template<class Engine,
    int P, int R>
    class discard_block {
public:
    typedef Engine base_type;
    typedef typename base_type::result_type result_type;
    static const int block_size = P;
    static const int used_block = R;
    discard_block();
    explicit discard_block(const base_type& eng);
    template<class Gen>
        discard_block(Gen& gen);
    void seed()
        {eng.seed(); }
    template<class Gen>
        void seed(Gen& gen)
```

```
        {eng.seed(gen); }
    const base_type& base() const;
    result_type min() const;
    result_type max() const;
    result_type operator()();
private:
    Engine stored_eng;       // exposition only
    int count;               // exposition only
    };
```

Constructor:

discard_block::discard_block:

```
discard_block()
    : count(0) {}
explicit discard_block(const base_type& eng)
    : stored_eng(eng), count(0) {}
template<class Gen>
    discard_block(Gen& gen)
    : stored_eng(gen), count(0) {}
```

The first constructor constructs a `discard_block` object with a default-initialized engine. The second constructor constructs a `discard_block` object with a copy of an engine object. The third constructor constructs a `discard_block` object with an engine initialized from a generator.

Member functions:

discard_block::base:

```
const base_type& base() const;
```

The member function returns a reference to the underlying engine object.

discard_block::operator():

```
result_type operator()()
    {
    if (R <= count)
        {
        while (count++ < P)
            stored_eng();
        count = 0;
        }
    ++count;
    return stored_eng();
    }
```

The member function returns the next value in the sequence.

discard_block::seed:

```
void seed()
    {stored_eng.seed(); count = 0; }
template<class Gen>
    void seed(Gen& gen)
    {stored_eng.seed(gen); count = 0; }
```

The first seed function calls `stored_eng.seed()` and sets `count` to 0. The second seed function calls `stored_eng.seed(gen)` and sets `count` to 0.

Types:

discard_block::base_type:

```
typedef Engine base_type;
```

The is a synonym for the type of the underlying engine object.

Constants:

discard_block::block_size:

```
static const int block_size = P;
```

The static const variable holds the value of the template argument `P`, the number of values in each cycle.

discard_block::used_block:

```
static const int block_size = R;
```

The static const variable holds the value of the template argument `R`, the number of values to return at the beginning of each cycle.

exponential_distribution

Description: The template class describes a distribution that produces values of a user-specified floating-point type with an exponential distribution.

Synopsis:

```
template<class RealType = double>
    class exponential_distribution {
public:
    typedef RealType input_type;
    typedef RealType result_type;
    explicit exponential_distribution(const result_type& lambda0 = result_type(1));
    result_type lambda() const;
    void reset();
    template<class Engine>
        result_type operator()(Engine& eng);
private:
    result_type stored_lambda;   // exposition only
    };
```

Constructor:

exponential_distribution::exponential_distribution:

```
exponential_distribution(const result_type& lambda0 = result_type(1));
```

Precondition: `0.0 <= lambda0`

The constructor constructs an object whose stored value `stored_lambda` holds the value `lambda0`.

Member functions:

exponential_distribution::lambda:

```
result_type lambda() const;
```

The member function returns the stored value `stored_lambda`.

exponential_distribution::operator():

```
template<class Engine>
result_type operator()(Engine& eng);
```

The member function uses the engine eng as a source of uniformly distributed random values and returns values with a probability density of pr(x) = stored_lambda * $e^{(-stored\_lambda * x)}$.

gamma_distribution

Description: The template class describes a distribution that produces values of a user-specified floating-point type with a gamma distribution.

Synopsis:

```
template<class RealType = double>
    class gamma_distribution {
public:
    typedef RealType input_type;
    typedef RealType result_type;
    explicit gamma_distribution(const result_type& alpha0 = result_type(1));
    result_type alpha() const;
    void reset();
    template<class Engine>
        result_type operator()(Engine& eng);
private:
    result_type stored_alpha;    // exposition only
    };
```

Constructor:

gamma_distribution::gamma_distribution:

```
gamma_distribution(const result_type& alpha0 = result_type(1));
```

Precondition: 0.0 < alpha0

The constructor constructs an object whose stored value stored_alpha holds the value alpha0.

Member functions:

gamma_distribution::alpha:

```
result_type alpha();
```

The member function returns the stored value stored_alpha.

gamma_distribution::operator():

```
template<class Engine>
result_type operator()(Engine& eng);
```

The member function uses the engine eng as a source of uniformly distributed random values and returns values with a probability density of pr(x) = 1 / gamma(stored_alpha) * $x^{stored\_alpha - 1}$ * e^{-x}.

geometric_distribution

Description: The template class describes a distribution that produces values of a user-specified integral type with a geometric distribution.

Synopsis:

```
template<class IntType = int, class RealType = double>
    class geometric_distribution {
public:
    typedef RealType input_type;
```

```
    typedef IntType result_type;
    explicit geometric_distribution(const RealType& p0 = RealType(0.5));
    RealType p() const;
    void reset();
    template<class Engine>
        result_type operator()(Engine& eng);
private:
    RealType stored_p;          // exposition only
    };
```

Constructor:

geometric_distribution::geometric_distribution:

```
geometric_distribution(const RealType& p0 = RealType(0.5));
```

Precondition: `0.0 < p0 && p0 < 1.0`

The constructor constructs an object whose stored value `stored_p` holds the value `p0`.

Member functions:

geometric_distribution::operator():

```
template<class Engine>
result_type operator()(Engine& eng);
```

The member function uses the engine `eng` as a source of uniformly distributed integral values and returns integral values with each value `i` occurring with probability $pr(i) = (1 - stored\_p) * stored\_p^{i - 1}$.

geometric_distribution::p:

```
RealType p() const;
```

The member function returns the stored value `stored_p`.

linear_congruential

Description: The template class describes a simple engine that produces values of a user-specified integral type using the **recurrence relation** `x(i) = (A * x(i-1) + C) mod M`. The engine's state is the last value returned, or the seed value if no call has been made to `operator()`.

The template argument UIntType must be large enough to hold values up to `M - 1`. The values of the template arguments `A` and `C` must be less than `M`.

Synopsis:

```
template<class UIntType,
    UIntType A, UIntType C, UIntType M>
    class linear_congruential {
public:
    typedef UIntType result_type;
    static const UIntType multiplier = A;
    static const UIntType increment = C;
    static const UIntType modulus = M;
    linear_congruential();
    explicit linear_congruential(unsigned long x0);
    template<class Gen>
        linear_congruential(Gen& gen);
    void seed(unsigned long x0 = 1);
    template<class Gen>
```

```
        void seed(Gen& gen);
    result_type min() const;
    result_type max() const;
    result_type operator()();
private:
    result_type stored_value;    // exposition only
    };
```

Constructor:

linear_congruential::linear_congruential:

```
linear_congruential();
explicit linear_congruential(unsigned long x0);
template<class Gen>
    linear_congruential(Gen& gen);
```

The first constructor constructs an object and initializes it by calling `seed()`. The second constructor constructs an object and initializes it by calling `seed(x0)`. The third constructor constructs an object and initializes it by calling `seed(gen)`.

Member functions:

linear_congruential::operator():

```
result_type operator()();
```

The member function generates a new `stored_value` by applying the recurrence relation to the old value of `stored_value`.

linear_congruential::seed:

```
void seed(unsigned long x0 = 1);
template<class Gen>
    void seed(Gen& gen);
```

The first seed function sets the stored value `stored_value` to `1` if `c mod M == 0` and `x0 mod M == 0`, otherwise it sets the stored value to `x0 mod M`. The second seed function calls `seed(gen())`.

Constants:

linear_congruential::increment:

```
static const UIntType increment = C;
```

The static const variable holds the value of the template argument `C`.

linear_congruential::modulus:

```
static const UIntType modulus = M;
```

The static const variable holds the value of the template argument `M`.

linear_congruential::multiplier:

```
static const UIntType multiplier = A;
```

The static const variable holds the value of template argument `A`.

mersenne_twister

Description: The template class describes a simple engine. It holds a large integral value with `W * (N - 1) + R` bits. It extracts `W` bits at a time from this large value, and when it has used all the bits it twists the large value by shifting and mixing the bits so that it has a new set of bits to extract from. The engine's state is the last N W-bit values used if operator() has been called at least N times, otherwise the M W-bit values that have been used and the last N - M values of the seed.

The template argument UIntType must be large enough to hold values up to 2^W - 1. The values of the other template arguments must satisfy the following requirements:

- 0 < M <= N
- 0 <= R, U, S, T, L <= W
- 0 <= A, B, C <= 2^W
- W * (N - 1) + R must be a Mersenne prime

The generator **twists** the large value that it holds by executing the following code:

```
for (int i = 0; i < N; ++i)
    {
    temp = (x[i] & LMASK) << (W - 1) | (x[i + 1] & HMASK) >> 1;
    if (temp & 1)
        y[i] = (temp >> 1) ^ A ^ x[(i + R) % N];
    else
        y[i] = (temp >> 1) ^ x[(i + R) % N];
    }
    for (int i = 0; i < N; ++i)
        x[i] = y[i];
```

where LMASK is an unsigned W-bit value with its low R bits set to 1 and the rest of its bits set to 0, and HMASK is the complement of LMASK.

The generator holds a current index idx initialized to 0. It extracts bits by executing the following code:

```
temp = x[idx++];
temp = temp ^ (temp >> U);
temp = temp ^ ((temp << S) & B);
temp = temp ^ ((temp << T) & C);
temp = temp ^ (temp >> L);
```

When idx reaches N the generator twists the stored value and sets idx back to 0.

Synopsis:

```
template<class UIntType,
    int W, int N, int R,
    UIntType A, int U, int S,
    UIntType B, int T, UIntType C, int L>
    class mersenne_twister {
public:
    typedef UIntType result_type;
    static const int word_size = W;
    static const int state_size = N;
    static const int shift_size = M;
    static const int mask_bits = R;
    static const int UIntType parameter_a = A;
    static const int output_u = U;
    static const int output_s = S;
    static const UIntType output_b = B;
    static const int output_t = T;
    static const UIntType output_c = C;
    static const int output_l = L;
    mersenne_twister();
    explicit mersenne_twister(unsigned long x0);
    template<class Gen>
```

```
        mersenne_twister(Gen& gen);
    void seed()
        {seed(5489); }
    void seed(unsigned long x0);
    template<class Gen>
        void seed(Gen& gen);
    result_type min() const;
    result_type max() const;
    result_type operator()();
    };
```

Constructor:

mersenne_twister::mersenne_twister:

```
mersenne_twister();
explicit mersenne_twister(unsigned long x0);
template<class Gen>
    mersenne_twister(Gen& gen);
```

The first constructor constructs an object and initializes it by calling `seed()`. The second constructor constructs an object and initializes it by calling `seed(x0)`. The third constructor constructs an object and initializes it by calling `seed(gen)`.

Member functions:

mersenne_twister::operator():

```
result_type operator()();
```

The member function extracts the next value in the sequence and returns it.

mersenne_twister::seed:

```
template<class Gen>
    void seed(Gen& gen);
void seed()
    {seed(5489); }
void seed(unsigned long x0);
```

Precondition: `0 < x0`

The first seed function generates `N` values from the values of type `unsigned long` returned by successive invocations of `gen` and then twists the resulting large integer value. Each value is `gen() % 2`^{W}.

The second seed function calls `seed(4357)`.

The third seed function sets the oldest historical value `h[0]` to `x0 mod 2`^{W}, then iteratively sets each successive historical value `h[i]` to `(i + 1812433253 * (h[i - 1] >> (W - 2))) mod 2`^{W}, for `i` ranging from `1` to `N - 1`.

Constants:

mersenne_twister::mask_bits:

```
static const int mask_bits = R;
```

The static const variable holds the value of the template argument `R`.

mersenne_twister::output_b:

```
static const UIntType output_b = B;
```

The static const variable holds the value of the template argument B.

mersenne_twister::output_c:

```
static const UIntType output_c = C;
```

The static const variable holds the value of the template argument C.

mersenne_twister::output_l:

```
static const int output_l = L;
```

The static const variable holds the value of the template argument L.

mersenne_twister::output_s:

```
static const int output_s = S;
```

The static const variable holds the value of the template argument S.

mersenne_twister::output_t:

```
static const int output_t = T;
```

The static const variable holds the value of the template argument T.

mersenne_twister::output_u:

```
static const int output_u = U;
```

The static const variable holds the value of the template argument U.

mersenne_twister::parameter_a:

```
static const int UIntType parameter_a = A;
```

The static const variable holds the value of the template argument A.

mersenne_twister::shift_size:

```
static const int shift_size = M;
```

The static const variable holds the value of the template argument M.

mersenne_twister::state_size:

```
static const int state_size = N;
```

The static const variable holds the value of the template argument N.

mersenne_twister::word_size:

```
static const int word_size = W;
```

The static const variable holds the value of the template argument W.

normal_distribution

Description: The template class describes a distribution that produces values of a user-specified floating-point type with a normal distribution.

Synopsis:

```
template<class RealType = double>
    class normal_distribution {
public:
```

```
    typedef RealType input_type;
    typedef RealType result_type;
    explicit normal_distribution(const result_type& mean0 = 0,
        const result_type& sigma0 = 1);
    result_type mean() const;
    result_type sigma() const;
    void reset();
    template<class Engine>
        result_type operator()(Engine& eng);
private:
    result_type stored_mean;     // exposition only
    result_type stored_sigma;    // exposition only
    };
```

Constructor:

normal_distribution::normal_distribution:

```
normal_distribution(const result_type& mean0 = 0,
    const result_type& sigma0 = 1);
```

Precondition: `0.0 <= sigma0`

The constructor constructs an object whose stored value `stored_mean` holds the value `mean0` and whose stored value `stored_sigma` holds the value `sigma0`.

Member functions:

normal_distribution::mean:

```
result_type mean() const;
```

The member function returns the stored value `stored_mean`.

normal_distribution::operator():

```
template<class Engine>
result_type operator()(Engine& eng);
```

The member function uses the engine `eng` as a source of uniformly distributed random values and returns returns values with a probability density of $\mathtt{pr(x) = 1 / ((2 * pi)^{1/2} * stored\_sigma) * e^{-(x - stored\_mean)^2 / (2 * stored\_sigma^2)}}$.

normal_distribution::sigma:

```
result_type sigma() const;
```

The member function returns the stored value `stored_sigma`.

poisson_distribution

Description: The template class describes a distribution that produces values of a user-specified integral type with a poisson distribution.

Synopsis:

```
template<class IntType = int, class RealType = double>
    class poisson_distribution {
public:
    typedef RealType input_type;
    typedef IntType result_type;
    explicit poisson_distribution(const RealType& mean0 = RealType(1));
    RealType mean() const;
    void reset();
    template<class Engine>
```

```
        result_type operator()(Engine& eng);
private:
    RealType stored_mean;     // exposition only
    };
```

Constructor:

poisson_distribution::poisson_distribution:

```
poisson_distribution(const RealType& mean0 = RealType(1));
```

Precondition: `0.0 < mean0`

The constructor constructs an object whose stored value `stored_mean` holds the value `mean0`.

Member functions:

poisson_distribution::mean:

```
RealType mean() const;
```

The member function returns the stored value `stored_mean`.

poisson_distribution::operator():

```
template<class Engine>
result_type operator()(Engine& eng);
```

The member function uses the engine `eng` as a source of uniformly distributed integral values and returns integral values with each value `i` occuring with probability $pr(i) = e^{-stored\_mean} * stored\_mean^{i} / i!$.

random_device

Description: The class describes a source of non-deterministic random numbers. In this implementation the values produced are not non-deterministic. They are uniformly distributed in the closed range `[0, 65535]`.

Synopsis:

```
class random_device {
public:
    typedef unsigned int result_type;
    explicit random_device(const std::string& token = /* implementation defined */);
    result_type min() const;
    result_type max() const;
    double entropy() const;
    result_type operator()();
private:
    random_device(const random_device&);    // exposition only
    void operator=(const random_device&);   // exposition only
    };
```

Constructor:

random_device::random_device:

```
random_device(const std::string& str);
```

The constructor initializes the device (as needed) with `str`.

Member functions:

random_device::entropy:

```
double entropy() const;
```

The member function returns an estimate of the randomness of the source, as measured in bits. (In the extreme, a non-random source has an entropy of zero.)

random_device::max:

```
result_type max() const;
```

The member function returns the largest value returned by the source.

random_device::min:

```
result_type min() const;
```

The member function returns the smallest value returned by the source.

random_device::operator():

```
result_type operator()();
```

The member function returns values uniformly distributed in the closed interval `[min(), max()]`.

Types:

random_device::result_type:

```
typedef unsigned int result_type;
```

The type is a synonym for `unsigned int`.

subtract_with_carry

Description: The template class describes a simple engine that produces values of a user-specified integral type using the **recurrence relation** `x(i) = (x(i - R) - x(i - S) - cy(i - 1)) mod M`, where `cy(i)` has the value `1` if `x(i - S) - x(i - R) - cy(i - 1) < 0`, otherwise `0`. The engine's state is the last `R` values returned if `operator()` has been called at least `R` times, otherwise the `M` values that have been returned and the last `R - M` values of the seed.

The template argument IntType must be large enough to hold values up to `M - 1`. The values of the template arguments `S` and `R` must be greater than `0` and `S` must be less than `R`.

Synopsis:

```
template<class IntType,
    IntType M, int S, int R>
    class subtract_with_carry {
public:
    typedef IntType result_type;
    static const IntType modulus = M;
    static const int short_lag = S;
    static const int long_lag = R;
    subtract_with_carry();
    explicit subtract_with_carry(unsigned long x0);
    template<class Gen>
        subtract_with_carry(Gen& gen);
    void seed(unsigned long x0 = 19780503UL);
    template<class Gen>
        void seed(Gen& gen);
```

```
    result_type min() const;
    result_type max() const;
    result_type operator()();
    };
```

Constructor:

subtract_with_carry::subtract_with_carry:

```
subtract_with_carry();
explicit subtract_with_carry(unsigned long x0);
template<class Gen>
    subtract_with_carry(Gen& gen);
```

The first constructor constructs an object and initializes it by calling `seed()`. The second constructor constructs an object and initializes it by calling `seed(x0)`. The third constructor constructs an object and initializes it by calling `seed(gen)`.

Member functions:

subtract_with_carry::operator():

```
result_type operator()();
```

The member function generates the next value in the pseudo-random sequence by applying the recurrence relation to the stored historical values, stores the generated value, and returns it.

Constants:

subtract_with_carry::long_lag:

```
static const int long_lag = R;
```

The static const variable holds the value of the template argument `R`.

subtract_with_carry::modulus:

```
static const IntType modulus = M;
```

The static const variable holds the value of the template argument `M`.

subtract_with_carry::short_lag:

```
static const int short_lag = S;
```

The static const variable holds the value of the template argument `S`.

subtract_with_carry::seed:

```
void seed(result_type x0 = 19780503UL);
template<class Gen>
    void seed(Gen& gen);
```

Precondition: `0 < x0`

The first seed function generates `long_lag` historical values from the values of type `unsigned long` returned by successive invocations of `gen`. Each historical value is `gen() % modulus`.

The second seed function effectively executes the following code:

```
    linear_congruential<unsigned long, 40014, 0, 2147483563> gen(x0);
    seed(gen);
```

subtract_with_carry_01

Description: The template class describes a simple engine that produces values of a user-specified floating-point type using the **recurrence relation** `x(i) = (x(i - R) - x(i - S) - cy(i - 1)) mod 1`, where `cy(i)` has the value 2^{-W} if `x(i - S) - x(i - R) - cy(i - 1) < 0`, otherwise `0`. The engine's state is the last `R` values returned if `operator()` has been called at least `R` times, otherwise the `M` values that have been returned and the last `R - M` values of the seed.

The template argument RealType must be large enough to hold values with `W` fraction bits. The values of the template arguments `S` and `R` must be greater than 0 and `S` must be less than `R`.

Synopsis:

```
template<class RealType,
    int W, int S, int R>
    class subtract_with_carry_01 {
public:
    typedef RealType result_type;
    static const int word_size = W;
    static const int short_lag = S;
    static const int long_lag = R;
    subtract_with_carry_01();
    explicit subtract_with_carry_01(unsigned long x0);
    template<class Gen>
        subtract_with_carry_01(Gen& gen);
    void seed(unsigned long x0 = 19780503UL);
    template<class Gen>
        void seed(Gen& gen);
    result_type min() const;
    result_type max() const;
    result_type operator()();
    };
```

Constructor:

subtract_with_carry_01::subtract_with_carry_01:

```
subtract_with_carry_01();
explicit subtract_with_carry_01(IntType x0);
template<class In>
    subtract_with_carry_01(InIt& first, InIt last);
```

The first constructor constructs an object and initializes it by calling `seed()`. The second constructor constructs an object and initializes it by calling `seed(x0)`. The third constructor constructs an object and initializes it by calling `seed(first, last)`.

Member functions:

subtract_with_carry_01::operator():

```
result_type operator()();
```

The member function generates the next value in the pseudo-random sequence by applying the recurrence relation to the stored historical values, stores the generated value, and returns it.

subtract_with_carry_01::seed:

```
template<class Gen>
    void seed(Gen& gen);
void seed(result_type x0 = 19780503UL);
```

Precondition: `0 < x0`

The first seed function generates `long_lag` historical values from the values of type `unsigned long` returned by successive invocations of `gen`. Each historical value is generated by concatenating the low 32 bits from each of `long_lag * (word_size + 31) / 32` values from the initialization sequence; the resulting value is then divided by 2.0^{word_size} and the integral part discarded. Thus, each historical value is a floating-point value greater than or equal to 0.0 and less than 1.0, with `word_size` significant bits.

The second seed function effectively executes the following code:

```
linear_congruential<unsigned long, 40014, 0, 2147483563> gen(x0);
seed(gen);
```

Constants:

subtract_with_carry_01::long_lag:

```
static const int long_lag = R;
```

The static const variable holds the value of the template argument `R`.

subtract_with_carry_01::short_lag:

```
static const int short_lag = S;
```

The static const variable holds the value of the template argument `S`.

subtract_with_carry_01::word_size:

```
static const int word_size = W;
```

The static const variable holds the value of the template argument `W`.

uniform_int

Description: The template class describes a distribution that produces values of a user-specified integral type with a uniform distribution.

Synopsis:

```
template<class IntType = int>
    class uniform_int {
public:
    typedef IntType input_type;
    tpyedef IntType result_type;
    explicit uniform_int(result_type min0 = 0, result_type max0 = 9);
    result_type min() const
        {return stored_min; }
    result_type max() const;
        {return stored_max; }
    void reset();
    template<class Engine>
        result_type operator()(Engine& eng);
    template<class Engine>
        result_type operator()(Engine& eng, result_type n);
private:
    result_type stored_min;    // exposition only
    result_type stored_max;    // exposition only
    };
```

Constructor:

uniform_int::uniform_int:

```
explicit uniform_int(result_type min0 = 0, result_type max0 = 9);
```

Precondition: `min0 < max0`

The constructor constructs an object whose stored value `stored_min` holds the value `min0` and whose stored value `stored_max` holds the value `max0`.

Member functions:

uniform_int::operator():

```
template<class Engine>
    result_type operator()(Engine& eng);
template<class Engine>
    result_type operator()(Engine& eng, result_type n);
```

The first member function uses the engine `eng` as a source of uniformly distributed integral values and returns integral values with each value `i` in the closed range `[min(), max()]` occurring with equal probability and values outside that range occurring with probability 0.

The second member function uses the engine `eng` as a source of uniformly distributed integral values and returns integral values with each value `i` in the closed range `[min(), n]` occurring with equal probability and values outside that range occurring with probability 0.

uniform_real

Description: The template class describes a distribution that produces values of a user-specified floating-point type with a uniform distribution.

Synopsis:

```
template<class RealType = double>
    class uniform_real {
public:
    typedef RealType input_type;
    tpyedef RealType result_type;

    explicit uniform_real(result_type min0 = result_type(0),
        result_type max0 = result_type(1));

    result_type min() const
        {return stored_min; }
    result_type max() const
        {return stored_max; }
    void reset();
    template<class Engine>
        result_type operator()(Engine& eng);
private:
    result_type stored_min;    // exposition only
    result_type stored_max;    // exposition only
    };
```

Constructor:

uniform_real::uniform_real:

```
explicit uniform_real(result_type min0 = result_type(0),
    result_type max0 = result_type(1));
```

Precondition: `min0 < max0`

The constructor constructs an object whose stored value `stored_min` holds the value `min0` and whose stored value `stored_max` holds the value `max0`.

Member functions:

uniform_real::operator():

```
template<class Engine>
bool operator()(Engine& eng);
```

The member function uses the engine `eng` as a source of uniformly distributed floating-point values and returns floating-point values with each value `x` in the half open range `[min(), max()]` occurring with equal probability and values outside that range occurring with probability 0.

variate_generator

Description: The template class describes an object that holds an engine and a distribution and produces values by passing the wrapped engine object to the `distribution` object's `operator()`.

The template argument `Engine` can be a type `Eng`, `Eng*`, or `Eng&`, where `Eng` is an engine. The type `Eng` is the **underlying engine type**. The corresponding object of type `Eng` is the the **underlying engine object**.

The template uses a **wrapped engine** to match the type of the values produced by the `engine` object to the type of values required by the `distribution` object. The wrapped engine's `operator()` returns values of type `Dist::input_type`, generated as follows:

- if `Engine::result_type` and `Dist::input_type` are both integral types it returns `eng()`, converted to type `Dist::input_type`.
- if `Engine::result_type` and `Dist::input_type` are both floating-point types it returns `(eng() - eng.min()) / (eng.max() - eng.min())`, converted to type `Dist::input_type`.
- if `Engine::result_type` is an integral type and `Dist::input_type` is a floating-point type it returns `(eng() - eng.min()) / (eng.max() - eng.min() + 1)`, converted to type `Dist::input_type`.
- if `Engine::result_type` is a floating-point type and `Dist::input_type` is an integral type it returns `((eng() - eng.min()) / (eng.max() - eng.min()) * std::numeric_limits<Dist::input_type>::max()`, converted to type `Dist::input_type`.

Synopsis:

```
template<class Engine, class Dist>
    class variate_generator {
public:
    typedef Engine engine_type;
    typedef engine-type engine_value_type;
    typedef Dist distribution_type;
    typedef typename Dist::result_type result_type;
    variate_generator(engine_type eng0, distribution_type dist0);
    result_type operator()();
    template<class T>
        result_type operator()(T value);
    engine_value_type& engine();
    const engine_value_type& engine() const;
    distribution_type& distribution();
    const distribution_type& distribution() const;
    result_type min() const
```

```
        {return dist.min(); }
    result_type max() const
        {return dist.max(); }
private:
    Engine eng;             // exposition only
    Dist dist;              // exposition only
    };
```

Constructor:

variate_generator::variate_generator:

```
variate_generator(engine_type eng0, distribution_type dist0);
```

The constructor constructs an object whose stored value `eng` holds `eng0` and whose stored value `dist` holds `dist0`.

Member functions:

variate_generator::distribution:

```
distribution_type& distribution();
const distribution_type& distribution() const;
```

The member functions return a reference to the stored distribution object `dist`.

variate_generator::engine:

```
engine_value_type engine();
const engine_value_type& engine() const;
```

The member functions return a reference to the underlying engine object.

variate_generator::max:

```
result_type max() const
    {return dist.max(); }
```

The member function returns `dist.max()`.

variate_generator::min:

```
result_type min() const
    {return dist.min(); }
```

The member function returns `dist.min()`.

variate_generator::operator():

```
result_type operator()();
template<class T>
    result_type operator()(T value);
```

The first member function returns `dist(wr_eng)`, where `wr_eng` is the object's wrapped engine.

The second member function returns `dist(wr_eng, value)`, where `wr_eng` is the object's wrapped engine.

Types:

variate_generator::distribution_type:

```
typedef Dist distribution_type;
```

The type is a synonym for the template parameter Dist.

variate_generator::engine_type:

```
typedef Engine engine_type;
```

The type is a synonym for the template parameter Engine.

variate_generator::engine_value_type:

```
typedef engine-type engine_value_type;
```

The type is a synonym for the underlying engine type.

variate_generator::result_type:

```
typedef typename Dist::result_type result_type;
```

The type is a synonym for Dist::result_type.

xor_combine

Description: The template class describes a compound engine that produces values by combining values produced by two engines. The engine's state is the state of stored_eng1 followed by the state of stored_eng2.

Synopsis:

```
template<class Engine1, int S1,
    class Engine2, int S2>
    class xor_combine {
public:
    typedef Engine1 base1_type;
    typedef Engine2 base2_type;
    typedef xxx result_type;
    static const int shift1 = S1;
    static const int shift2 = S2;
    xor_combine()
        {}
    xor_combine(const base1_type& eng1, const base2_type& eng2)
        : stored_eng1(eng1), stored_eng2(eng2) {}
    template<class Gen>
        xor_combine(Gen& gen)
        : stored_eng1(gen), stored_eng2(gen) {}
    void seed()
        {stored_eng1.seed(); stored_eng2.seed(); }
    template<class Gen>
        void seed(Gen& gen);
        {stored_eng1.seed(gen); stored_eng2.seed(gen); }
    const base1_type& base1() const;
    const base2_type& base2() const;
    result_type min() const;
    result_type max() const;
    result_type operator()();
private:
    base1_type stored_eng1;     // exposition only
    base2_type stored_eng2;     // exposition only
    };
```

Constructor:

xor_combine::xor_combine:

```
xor_combine()
    {}
xor_combine(const base1_type& eng1, const base2_type& eng2);
```

```
    : stored_eng1(eng1), stored_eng2(eng2)
    {}
template<class Gen>
    xor_combine(Gen& gen)
    : stored_eng1(gen), stored_eng2(gen)
    {}
```

The first constructor constructs an object with stored values `stored_eng1` and `stored_eng2` constructed with their respective default constructors. The second constructor constructs an object whose stored value `stored_eng1` holds a copy of `eng1` and whose stored value `stored_eng2` holds a copy of `eng2`. The third constructor constructs an object and calls `seed(gen)`.

Member functions:

xor_combine::base1:

```
const base1_type& base1() const;
```

The member function returns a reference to the stored value `stored_eng1`.

xor_combine::base2:

```
const base2_type& base2() const;
```

The member function returns a reference to the stored value `stored_eng2`.

xor_combine::seed:

```
void seed()
    {stored_eng1.seed(); stored_eng2.seed(); }
template<class Gen>
    void seed(Gen& gen);
    {stord_eng1.seed(gen); stored_eng2.seed(gen); }
```

The first member function calls `stored_eng1.seed()` and then calls `stored_eng2.seed()`. The second member function calls `stored_eng1.seed(gen)` and then calls `stored_eng2.seed(gen)`.

Types:

xor_combine::base1_type:

```
typedef Engine1 base1_type;
```

The type names the template parameter `Engine1`.

xor_combine::base2_type:

```
typedef Engine1 base2_type;
```

The type names the template parameter `Engine2`.

Constants:

xor_combine::shift1:

```
static const int shift1 = S1;
```

The static const variable holds the value of the template argument `S1`.

xor_combine::shift2:

```
static const int shift2 = S2;
```

The static const variable holds the value of the template argument S2.

Operators:

operator():

```
result_type operator()();
```

The member operator returns (stored_eng1() << shift1) ^ (stored_eng2() << shift2).

Types

minstd_rand0

```
typedef linear_congruential< i-type, 16807, 0, 2147483647> minstd_rand0;
```

The type is a synonym for a specialization of the template linear_congruential.

minstd_rand

```
typedef linear_congruential< i-type, 48271, 0, 2147483647> minstd_rand;
```

The type is a synonym for a specialization of the template linear_congruential.

mt19937

```
typedef mersenne_twister< ui-type, 32, 624, 397, 31,
    0x9908b0df, 11, 7, 0x9d2c5680, 15, 0xefc60000, 18> mt19937;
```

The type is a synonym for a specialization of the template mersenne_twister.

ranlux_base_01

```
typedef subtract_with_carry_01<float, 24, 10, 24> ranlux_base_01;
```

The type is a synonym for a specialization of the template subtract_with_carry_01.

ranlux3

```
typedef discard_block<subtract_with_carry< i-type,
1 << 24, 10, 24>, 223, 24> ranlux3;
```

The type is a synonym for a specialization of the template discard_block with a specialization of the template subtract_with_carry.

ranlux3_01

```
typedef discard_block<subtract_with_carry_01<float, 24, 10, 24>,
223, 24> ranlux3_01;
```

The type is a synonym for a specialization of the template discard_block with a specialization of the template subtract_with_carry_01.

ranlux4

```
typedef discard_block<subtract_with_carry< i-type, 1 << 24, 10, 24>,
389, 24> ranlux4;
```

The type is a synonym for a specialization of the template discard_block with a specialization of the template subtract_with_carry.

ranlux4_01

```
typedef discard_block<subtract_with_carry_01<float, 24, 10, 24>,
389, 24> ranlux4_01;
```

The type is a synonym for a specialization of the template `discard_block` with a specialization of the template `subtract_with_carry_01`.

ranlux64_base_01

```
typedef subtract_with_carry_01<double, 48, 10, 24> ranlux64_base_01;
```

The type is a synonym for a specialization of the template `subtract_with_carry_01`.

<regex>

Description

Include the TR1 header **<regex>** to define a template class to parse regular expressions and several template classes and functions to search text for matches to a regular expression object.

Note: To enable this header file, you must define the macro __IBMCPP_TR1__ and use the **TARGET** compiler option to specify a valid release level. Valid release levels for TR1 support are zOSV1R12 or later. For example, you can specify the **TARGET** option as follows: `TARGET(zOSV1R12)`.

Any release prior to zOSV1R12 is invalid for the use of TR1. If TR1 code is used in an application to be compiled on an earlier platform, the compiler will issue the `#error` directive.

To **create** a regular expression object use the template class "basic_regex" on page 300 or one of its specializations, "regex" on page 322 and "wregex" on page 325, along with the syntax flags of type syntax_option_type.

To **search** text for matches to a regular expression object use the template functions "regex_match" on page 319 and "regex_search" on page 320, along with the match flags of type match_flag_type. These functions return their results using the template class "match_results" on page 304 and its specializations, "cmatch" on page 321, "wcmatch" on page 324, "smatch" on page 323, and "wsmatch" on page 324, along with the template class "sub_match" on page 317 and its specializations, "csub_match" on page 321, "wcsub_match" on page 324, "ssub_match" on page 324, and "wssub_match" on page 325.

To **replace** text that matches a regular expression object use the template function "regex_replace" on page 320, along with the match flags of type match_flag_type.

To **iterate** through multiple matches of a regular expression object use the template classes "regex_iterator" on page 309 and "regex_token_iterator" on page 311 or one of their specializations, "cregex_iterator" on page 321, "sregex_iterator" on page 324, "wcregex_iterator" on page 324, "wsregex_iterator" on page 324, "cregex_token_iterator" on page 321, "sregex_token_iterator" on page 324, "wcregex_token_iterator" on page 324, and "wsregex_token_iterator" on page 324, along with the match flags of type match_flag_type.

To **modify** some of the details of the grammar of regular expressions write a class that implements the regular expression traits.

Synopsis

```
namespace std {
    namespace tr1 {

 // TEMPLATE CLASS regex_traits AND basic_regex
 template<class Elem>
    struct regex_traits;
 template<>
    struct regex_traits<char>;
 template<>
    struct regex_traits<wchar_t>;
 template<class Elem,
    class RXtraits = regex_traits<Elem>,
    class basic_regex;
 typedef basic_regex<char> regex;
 typedef basic_regex<wchar_t> wregex;

 // TEMPLATE CLASS sub_match
 template<class BidIt>
    class sub_match;
 typedef sub_match<const char*> csub_match;
 typedef sub_match<const wchar_t*> wcsub_match;
 typedef sub_match<string::const_iterator> ssub_match;
 typedef sub_match<wstring::const_iterator> wssub_match;

 // TEMPLATE CLASS match_results
 template<class BidIt,
    class Alloc = allocator<typename iterator_traits<BidIt>::value_type> >
    class match_results;
 typedef match_results<const char*> cmatch;
 typedef match_results<const wchar_t*> wcmatch;
 typedef match_results<string::const_iterator> smatch;
 typedef match_results<wstring::const_iterator> wsmatch;

  // NAMESPACE regex_constants
        namespace regex_constants {
  typedef T1 syntax_option_type;
  static const syntax_option_type awk, basic, collate, ECMAScript,
    egrep, extended, grep, icase, nosubs, optimize;
  typedef T2 match_flag_type;
  static const match_flag_type match_any, match_default, match_not_bol,
    match_not_bow, match_continuous, match_not_eol, match_not_eow,
    match_not_null, match_partial, match_prev_avail;
  typedef T3 error_type;
  static const error_type error_badbrace, error_badrepeat, error_brace,
    error_brack, error_collate, error_complexity, error_ctype,
    error_escape, error_paren, error_range, error_space,
    error_stack, error_backref;
        }  // namespace regex_constants

 // CLASS regex_error
 class regex_error;

 // TEMPLATE FUNCTION regex_match
 template<class BidIt, class Alloc, class Elem, class RXtraits>
    bool regex_match(BidIt first, Bidit last,
        match_results<BidIt, Alloc>& match,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);
 template<class BidIt, class Elem, class RXtraits>
    bool regex_match(BidIt first, Bidit last,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);

 template<class Elem, class Alloc, class RXtraits>
    bool regex_match(const Elem* ptr,
```

```
        match_results<const Elem*, Alloc>& match,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);
template<class Elem, class RXtraits>
   bool regex_match(const Elem* ptr,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);

template<class IOtraits, class IOalloc, class Alloc, class Elem, class RXtraits>
   bool regex_match(
        const basic_string<Elem, IOtraits, IOalloc>& str,
        match_results<typename basic_string<Elem, IOtraits, IOalloc>::
        const_iterator, Alloc>& match,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags= match_default);
template<class IOtraits, class IOalloc, class Elem, class RXtraits>
   bool regex_match(
        const basic_string<Elem, IOtraits, IOalloc>& match,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);

// TEMPLATE FUNCTION regex_search
template<class BidIt, class Alloc, class Elem, class RXtraits>
   bool regex_search(BidIt first, Bidit last,
        match_results<BidIt, Alloc>& match,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);
template<class BidIt, class Elem, class RXtraits>
   bool regex_search(BidIt first, Bidit last,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);

template<class Elem, class Alloc, class RXtraits>
   bool regex_search(const Elem* ptr,
        match_results<const Elem*, Alloc>& match,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);
template<class Elem, class RXtraits>
   bool regex_search(const Elem* ptr,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);

template<class IOtraits, class IOalloc, class Alloc, class Elem, class RXtraits>
   bool regex_search(
        const basic_string<Elem, IOtraits, IOalloc>& str,
        match_results<typename basic_string<Elem, IOtraits, IOalloc>::
        const_iterator, Alloc>& match,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);
template<class IOtraits, class IOalloc, class Elem, class RXtraits>
   bool regex_search(
        const basic_string<Elem, IOtraits, IOalloc>& str,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);

// TEMPLATE FUNCTION regex_replace
template<class OutIt, class BidIt, class RXtraits, class Elem>
   OutIt regex_replace(OutIt out, BidIt first, BidIt last,
        const basic_regex<Elem, RXtraits>& re,
        const basic_string<Elem>& fmt,
        match_flag_type flags = match_default);
template<class RXtraits, class Elem>
   basic_string<Elem> regex_replace(
        const basic_string<Elem>& str,
        const basic_regex<Elem, RXtraits>& re,
        const basic_string<Elem>& fmt,
        match_flag_type flags = match_default);
```

```
// REGULAR EXPRESSION ITERATORS
template<class BidIt, class Elem = iterator_traits<BidIt>::value_type,
   class RXtraits = regex_traits<Elem> >
       class regex_iterator;
typedef regex_iterator<const char*> cregex_iterator;
typedef regex_iterator<const wchar_t*> wcregex_iterator;
typedef regex_iterator<string::const_iterator> sregex_iterator;
typedef regex_iterator<wstring::const_iterator> wsregex_iterator;

template<class BidIt, class Elem = iterator_traits<BidIt>::value_type,
   class RXtraits = regex_traits<Elem> >
       class regex_token_iterator;
typedef regex_token_iterator<const char*> cregex_token_iterator;
typedef regex_token_iterator<const wchar_t*> wcregex_token_iterator;
typedef regex_token_iterator<string::const_iterator> sregex_token_iterator;
typedef regex_token_iterator<wstring::const_iterator> wsregex_token_iterator;

// STREAM INSERTER
template<class Elem, class IOtraits, class Alloc, class BidIt>
   basic_ostream<Elem, IOtraits>&
   operator<<(
       basic_ostream<Elem, IOtraits>& os,
       const sub_match<BidIt>& submatch);

// TEMPLATE swap FUNCTIONS
template<class Elem, class RXtraits>
   void swap(
       basic_regex<Elem, RXtraits>& left,
       basic_regex<Elem, RXtraits>& right) throw();
template<class Elem, class IOtraits, class BidIt, class Alloc>
   void swap(
       match_results<BidIt, Alloc>& left,
       match_results<BidIt, Alloc>& right) throw();

// COMPARISON OPERATORS FOR match_results
template<class BidIt, class Alloc>
   bool operator==(
       const match_results<BidIt, Alloc>& left,
       const match_results<BidIt, Alloc>& right);
template<class BidIt, class Alloc>
   bool operator!=(
       const match_results<BidIt, Alloc>& left,
       const match_results<BidIt, Alloc>& right);

// COMPARISON OPERATORS FOR sub_match
template<class BidIt>
   bool operator==(
       const sub_match<BidIt>& left,
       const sub_match<BidIt>& right);
template<class BidIt>
   bool operator!=(
       const sub_match<BidIt>& left,
       const sub_match<BidIt>& right);
template<class BidIt>
   bool operator<(
       const sub_match<BidIt>& left,
       const sub_match<BidIt>& right);
template<class BidIt>
   bool operator<=(
       const sub_match<BidIt>& left,
       const sub_match<BidIt>& right);
template<class BidIt>
   bool operator>(
       const sub_match<BidIt>& left,
       const sub_match<BidIt>& right);
template<class BidIt>
```

```
   bool operator>=(
       const sub_match<BidIt>& left,
       const sub_match<BidIt>& right);

template<class BidIt, class IOtraits, class Alloc>
   bool operator==(
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& left, const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
   bool operator!=(
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& left, const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
   bool operator<(
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& left, const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
   bool operator<=(
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& left, const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
   bool operator>(
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& left, const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
   bool operator>=(
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& left, const sub_match<BidIt>& right);

template<class BidIt, class IOtraits, class Alloc>
   bool operator==(
       const sub_match<BidIt>& left,
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& right);
template<class BidIt, class IOtraits, class Alloc>
   bool operator!=(
       const sub_match<BidIt>& left,
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& right);
template<class BidIt, class IOtraits, class Alloc>
   bool operator<(
       const sub_match<BidIt>& left,
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& right);
template<class BidIt, class IOtraits, class Alloc>
   bool operator<=(
       const sub_match<BidIt>& left,
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& right);
template<class BidIt, class IOtraits, class Alloc>
   bool operator>(
       const sub_match<BidIt>& left,
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& right);
template<class BidIt, class IOtraits, class Alloc>
   bool operator>=(
       const sub_match<BidIt>& left,
       const basic_string<typename iterator_traits<BidIt>::value_type,
       IOtraits, Alloc>& right);

template<class BidIt>
   bool operator==(
       const typename iterator_traits<BidIt>::value_type* left,
       const sub_match<BidIt>& right);
template<class BidIt>
   bool operator!=(
       const typename iterator_traits<BidIt>::value_type* left,
```

```
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator<(
        const typename iterator_traits<BidIt>::value_type* left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator<=(
        const typename iterator_traits<BidIt>::value_type* left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator>(
        const typename iterator_traits<BidIt>::value_type* left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator>=(
        const typename iterator_traits<BidIt>::value_type* left,
        const sub_match<BidIt>& right);

template<class BidIt>
    bool operator==(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type*);
template<class BidIt>
    bool operator!=(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type*);
template<class BidIt>
    bool operator<(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type*);
template<class BidIt>
    bool operator<=(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type*);
template<class BidIt>
    bool operator>(
        const sub_match<BidIt>&, left
        const typename iterator_traits<BidIt>::value_type*);
template<class BidIt>
    bool operator>=(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type*);

template<class BidIt>
    bool operator==(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator!=(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator<(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator<=(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator>(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator>=(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
```

```
template<class BidIt>
   bool operator==(
       const sub_match<BidIt>& left,
       const typename iterator_traits<BidIt>::value_type& right);
template<class BidIt>
   bool operator!=(
       const sub_match<BidIt>& left,
       const typename iterator_traits<BidIt>::value_type& right);
template<class BidIt>
   bool operator<(
       const sub_match<BidIt>& left,
       const typename iterator_traits<BidIt>::value_type& right);
template<class BidIt>
   bool operator<=(
       const sub_match<BidIt>& left,
       const typename iterator_traits<BidIt>::value_type& right);
template<class BidIt>
   bool operator>(
       const sub_match<BidIt>& left,
       const typename iterator_traits<BidIt>::value_type& right);
template<class BidIt>
   bool operator>=(
       const sub_match<BidIt>& left,
       const typename iterator_traits<BidIt>::value_type& right);
   }  // namespace tr1
}  // namespace std
```

Classes

basic_regex

Description: The template class describes an object that holds a regular expression. Objects of this template class can be passed to the template functions "regex_match" on page 319, "regex_search" on page 320, and "regex_replace" on page 320, along with suitable text string arguments, to search for text that matches the regular expression. The TR1 library provides two specializations of this template class, with the type definitions "regex" on page 322 for elements of type *char*, and "wregex" on page 325 for elements of type *wchar_t*.

The template argument `RXtraits` describes various important properties of the syntax of the regular expressions that the template class supports. A class that specifies these regular expression traits must have the same external interface as an object of template class "regex_traits" on page 314.

Some functions take an **operand sequence** that defines a regular expression. You can specify such an operand sequence several ways:

- `ptr` — a null-terminated sequence (such as a C string, for `Elem` of type *char*) beginning at `ptr` (which must not be a null pointer), where the terminating element is the value `value_type()` and is not part of the operand sequence
- `ptr`, `count` — a sequence of `count` elements beginning at `ptr` (which must not be a null pointer)
- `str` — the sequence specified by the `basic_string` object `str`
- `first`, `last` — a sequence of elements delimited by the iterators `first` and `last`, in the range `[first, last)`
- `right` — the `basic_regex` object `right`

These member functions also take an argument flags that specifies various options for the interpretation of the regular expression in addition to those described by the RXtraits type.

Synopsis:

```
template<class Elem,
    class RXtraits = regex_traits<Elem>,
    class basic_regex {
public:
    basic_regex();
    explicit basic_regex(const Elem *ptr,
        flag_type flags = ECMAScript);
    basic_regex(const Elem *ptr, size_type len,
        flag_type flags = ECMAScript);
    basic_regex(const basic_regex& right);
    template<class STtraits, class STalloc>
        explicit basic_regex(const basic_string<Elem, STtraits, STalloc>& str,
            flag_type flags = ECMAScript);
    template<class InIt>
        explicit basic_regex(InIt first, InIt last,
            flag_type flags = ECMAScript);

    basic_regex& operator=(const basic_regex& right);
    basic_regex& operator=(const Elem *ptr);
    template<class STtraits, class STalloc>
        basic_regex& operator=(const basic_string<Elem, STtraits, STalloc>& str);
    basic_regex& assign(const basic_regex& right);
    basic_regex& assign(const Elem *ptr,
        flag_type flags = ECMAScript);
    basic_regex& assign(const Elem *ptr, size_type len,
        flag_type flags = ECMAScript);
    template<class STtraits, class STalloc>
    basic_regex& assign(const basic_string<Elem, STtraits, STalloc>& str,
        flag_type flags = ECMAScript);
    template<class InIt>
        basic_regex& assign(InIt first, InIt last,
            flag_type flags = ECMAScript);

    locale_type imbue(locale_type loc);
    locale_type getloc() const;
    void swap(basic_regex& other) throw();

    unsigned mark_count() const;

    flag_type flags() const;

    typedef Elem value_type;
    typedef regex_constants::syntax_option_type flag_type;
    typedef typename RXtraits::locale_type locale_type;

    static const flag_type icase = regex_constants::icase;
    static const flag_type nosubs = regex_constants::nosubs;
    static const flag_type optimize = regex_constants::optimize;
    static const flag_type collate = regex_constants::collate;

    static const flag_type ECMAScript = regex_constants::ECMAScript;
    static const flag_type basic = regex_constants::basic;
    static const flag_type extended = regex_constants::extended;
    static const flag_type awk = regex_constants::awk;
    static const flag_type grep = regex_constants::grep;
    static const flag_type egrep = regex_constants::egrep;
private:
    RXtraits traits;    // exposition only
    };
```

Constructor:

basic_regex::basic_regex:

```
basic_regex();
template<class STtraits, class STalloc>
explicit basic_regex(const basic_string<Elem, STtraits, STalloc>& str,
    flag_type flags = ECMAScript);
template<class InIt>
explicit basic_regex(InIt first, InIt last,
    flag_type flags = ECMAScript);
explicit basic_regex(const Elem *ptr,
    flag_type flags = ECMAScript);
explicit basic_regex(const Elem *ptr, size_type len,
    flag_type flags);
basic_regex(const basic_regex& right);
```

All constructors store a default-constructed object of type `RXtraits`.

The first constructor constructs an empty `basic_regex` object. The other constructors construct a `basic_regex` object that holds the regular expression described by the operand sequence.

An **empty** `basic_regex` object does not match any character sequence when passed to "regex_match" on page 319, "regex_search" on page 320, or "regex_replace" on page 320.

Types:

basic_regex::flag_type:

```
typedef regex_constants::syntax_option_type flag_type;
```

The type is a synonym for regex_constants::syntax_option_type.

basic_regex::locale_type:

```
typedef typename RXtraits::locale_type locale_type;
```

The type is a synonym for regex_traits::locale_type.

basic_regex::value_type:

```
typedef Elem value_type;
```

The type is a synonym for the template parameter `Elem`.

Member functions:

basic_regex::assign:

```
basic_regex& assign(const basic_regex& right);
basic_regex& assign(const Elem *ptr,
    flag_type flags = ECMAScript);
basic_regex& assign(const Elem *ptr, size_type len,
    flag_type flags = ECMAScript);
template<class STtraits, class STalloc>
basic_regex& assign(const basic_string<Elem, STtraits, STalloc>& str,
    flag_type flags = ECMAScript);
template<class InIt>
basic_regex& assign(InIt first, InIt last,
    flag_type flags = ECMAScript);
```

The member functions each replace the regular expression held by `*this` with the regular expression described by the operand sequence, then return `*this`.

basic_regex::flags:

```
flag_type flags() const;
```

The member function returns the value of the `flag_type` argument passed to the most recent call to one of the assign member functions or, if no such call has been made, the value passed to the constructor.

basic_regex::getloc:

```
locale_type getloc() const;
```

The member function returns `traits.getloc()`.

basic_regex::imbue:

```
locale_type imbue(locale_type loc);
```

The member function empties `*this` and returns `traits.imbue(loc)`.

basic_regex::mark_count:

```
unsigned mark_count() const;
```

The member function returns the number of capture groups in the regular expression.

basic_regex::swap:

```
void swap(basic_regex& right) throw();
```

The member function swaps the regular expressions between `*this` and `right`. It does so in constant time and throws no exceptions.

Constants:

basic_regex::awk:

```
static const flag_type awk = regex_constants::awk;
```

The constant can be passed to the constructors or the assign member functions. It has the value regex_constants::awk.

basic_regex::basic:

```
static const flag_type basic = regex_constants::basic;
```

The constant can be passed to the constructors or the assign member functions. It has the value regex_constants::basic.

basic_regex::collate:

```
static const flag_type collate = regex_constants::collate;
```

The constant can be passed to the constructors or the assign member functions. It has the value regex_constants::collate.

basic_regex::ECMAScript:

```
static const flag_type ECMAScript = regex_constants::ECMAScript;
```

The constant can be passed to the constructors or the assign member functions. It has the value regex_constants::ECMAScript.

basic_regex::egrep:

```
static const flag_type egrep = regex_constants::egrep;
```

The constant can be passed to the constructors or the assign member functions. It has the value regex_constants::egrep.

basic_regex::extended:

```
static const flag_type extended = regex_constants::extended;
```

The constant can be passed to the constructors or the assign member functions. It has the value regex_constants::extended.

basic_regex::grep:

```
static const flag_type grep = regex_constants::grep;
```

The constant can be passed to the constructors or the assign member functions. It has the value regex_constants::grep.

basic_regex::icase:

```
static const flag_type icase = regex_constants::icase;
```

The constant can be passed to the constructors or the assign member functions. It has the value regex_constants::icase.

basic_regex::nosubs:

```
static const flag_type nosubs = regex_constants::nosubs;
```

The constant can be passed to the constructors or the assign member functions. It has the value regex_constants::nosubs.

basic_regex::optimize:

```
static const flag_type optimize = regex_constants::optimize;
```

The constant can be passed to the constructors or the assign member functions. It has the value regex_constants::optimize.

Operators:

basic_regex::operator=:

```
basic_regex& operator=(const basic_regex& right);
basic_regex& operator=(const Elem *str);
template<class STtraits, class STalloc>
basic_regex& operator=(const basic_string<Elem, STtraits, STalloc>& str);
```

The operators each replace the regular expression held by `*this` with the regular expression described by the operand sequence, then return `*this`.

match_results

Description: The template class describes an object that controls a non-modifiable sequence of elements of type `sub_match<BidIt>` generated by a regular expression search. Each element points to the subsequence that matched the capture group corresponding to that element.

Synopsis:

```
template<class BidIt,
    class Alloc = allocator<typename iterator_traits<BidIt>::value_type> >
    class match_results {
public:
    explicit match_results(const Alloc& alloc = Alloc());
    match_results(const match_results& right);

    match_results& operator=(const match_results& right);

    difference_type position(size_type sub = 0) const;
    difference_type length(size_type sub = 0) const;
    string_type str(size_type sub = 0) const;
    const_reference operator[](size_type n) const;

    const_reference prefix() const;
    const_reference suffix() const;
    const_iterator begin() const;
    const_iterator end() const;

    template<class OutIt>
        OutIt format(OutIt out,
            const string_type& fmt,
            match_flag_type flags = format_default) const;
    string_type format(const string_type& fmt,
        match_flag_type flags = format_default) const;

    allocator_type get_allocator() const;
    void swap(const match_results& other) throw();

    size_type size() const;
    size_type max_size() const;
    bool empty() const;

    typedef sub_match<BidIt> value_type;
    typedef const typename Alloc::const_reference const_reference;
    typedef const_reference reference;
    typedef T0 const_iterator;
    typedef const_iterator iterator;
    typedef typename iterator_traits<BidIt>::difference_type difference_type;
    typedef typename Alloc::size_type size_type;
    typedef Alloc allocator_type;
    typedef typename iterator_traits<BidIt>::value_type char_type;
    typedef basic_string<char_type> string_type;
    };
```

Constructor:

match_results::match_results:

```
explicit match_results(const Alloc& alloc = Alloc());
match_results(const match_results& right);
```

The first constructor constructs a `match_results` object that holds no submatches. The second constructor constructs a `match_results` object that is a copy of `right`.

Types:

match_results::allocator_type:

```
typedef Alloc allocator_type;
```

The typedef is a synonym for the template argument `Alloc`.

match_results::char_type:

```
typedef typename iterator_traits<BidIt>::value_type char_type;
```

The typedef is a synonym for the type iterator_traits<BidIt>::value_type, which is the element type of the character sequence that was searched.

match_results::const_iterator:

```
typedef T0 const_iterator;
```

The typedef describes an object that can server as a constant random-access iterator for the controlled sequence.

match_results::const_reference:

```
typedef const typename Alloc::const_reference const_reference;
```

The typedef describes an object that can serve as a constant reference to an element of the controlled sequence.

match_results::difference_type:

```
typedef typename iterator_traits<BidIt>::difference_type difference_type;
```

The typedef is a synonym for the type iterator_traits<BidIt>::difference_type; it describes an object that can represent the difference between any two iterators that point at elements of the controlled sequence.

match_results::iterator:

```
typedef const_iterator iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence.

match_results::reference:

```
typedef const_reference reference;
```

The type is a synonym for the type const_reference.

match_results::size_type:

```
typedef typename Alloc::size_type size_type;
```

The type is a synonym for the type Alloc::size_type.

match_results::string_type:

```
typedef basic_string<char_type> string_type;
```

The type is a synonym for the type basic_string<char_type>.

match_results::value_type:

```
typedef sub_match<BidIt> value_type;
```

The typedef is a synonym for the type sub_match<BidIt>.

Member functions:

match_results::begin:

```
const_iterator begin() const;
```

The member function returns a random access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

match_results::empty:

```
bool empty() const;
```

The member function returns true only if the regular expression search failed.

match_results::end:

```
const_iterator end() const;
```

The member function returns an iterator that points just beyond the end of the sequence.

match_results::format:

```
template<class OutIt>
    OutIt format(OutIt out,
        const string_type& fmt,
        match_flag_type flags = format_default) const;

string_type format(const string_type& fmt,
    match_flag_type flags = format_default) const;
```

Each member function generates formatted text under the control of the format `fmt`. The first member function writes the formatted text to the sequence defined by its argument `out` and returns `out`. The second member function returns a string object holding a copy of the formatted text.

To generate **formatted text** literal text in the format string is ordinarily copied to the target sequence. Each escape sequence in the format string is replaced by the text that it represents. The details of the copying and replacement are controlled by the format flags passed to the function.

match_results::get_allocator:

```
allocator_type get_allocator() const;
```

The member function returns a copy of the allocator object used by `*this` to allocate its `sub_match` objects.

match_results::length:

```
difference_type length(size_type sub = 0) const;
```

The member function returns `(*this)[sub].length()`.

match_results::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

match_results::operator[]:

```
const_reference operator[](size_type n) const;
```

The member function returns a reference to element `n` of the controlled sequence, or a reference to an empty `sub_match` object if `size() <= n` or if the capture group `n` was not part of the match.

match_results::position:

```
difference_type position(size_type sub = 0) const;
```

The member function returns `std::distance(prefix().first, (*this)[sub].first)`, that is, the distance from the first character in the target sequence to the first character in the submatch pointed to by element `n` of the controlled sequence.

match_results::prefix:

```
const_reference prefix() const;
```

The member function returns a reference to an object of type `sub_match<BidIt>` that points to the character sequence that begins at the start of the target sequence and ends at `(*this)[0].first`, that is, it points to the text that precedes the matched subsequence.

match_results::size:

```
size_type size() const;
```

The member function returns one more than the number of capture groups in the regular expression that was used for the search, or 0 if no search has been made.

match_results::str:

```
string_type str(size_type sub = 0) const;
```

The member function returns `string_type((*this)[sub])`.

match_results::suffix:

```
const_reference suffix() const;
```

The member function returns a reference to an object of type `sub_match<BidIt>` that points to the character sequence that begins at `(*this)[size() - 1].second` and ends at the end of the target sequence, that is, it points to the text that follows the matched subsequence.

match_results::swap:

```
void swap(const match_results& right) throw();
```

The member function swaps the contents of `*this` and `right` in constant time and does not throw exceptions.

Operators:

match_results::operator=:

```
match_results& operator=(const match_results& right);
```

The member operator replaces the sequence controlled by `*this` with a copy of the sequence controlled by `right`.

regex_error

Description: The class describes an exception object thrown to report an error in the construction or use of a `basic_regex` object.

Synopsis:

```
class regex_error : public std::runtime_error {
public:
    explicit regex_error(regex_constants::error_code error);
    regex_constants::error_code code() const;
    };
```

Constructor:

regex_error::regex_error:

```
regex_error(regex_constants::error_code error);
```

The constructor constructs an object that holds the value `error`.

Member functions:

regex_error::code:

```
regex_constants::error_code code() const;
```

The member function returns the value that was passed to the object's constructor.

regex_iterator

Description: The template class describes a constant forward iterator object. It extracts objects of type `match_results<BidIt>` by repeatedly applying its regular expression object `*pregex` to the character sequence defined by the iterator range `[begin,end)`.

Synopsis:

```
template<class BidIt, class Elem = iterator_traits<BidIt>::value_type,
    class RXtraits = regex_traits<Elem> >
        class regex_iterator {
public:
    typedef basic_regex<Elem, RXtraits> regex_type;
    typedef match_results<BidIt> value_type;
    typedef std::forward_iterator_tag iterator_category;
    typedef std::ptrdiff_t difference_type;
    typedef const match_results<BidIt>* pointer;
    typedef const match_results<BidIt>& reference;

    regex_iterator();
    regex_iterator(BidIt first, BidIt last, const regex_type& re,
   regex_constants::match_flag_type f = regex_constants::match_default);

    bool operator==(const regex_iterator& right);
    bool operator!=(const regex_iterator& right);

    const match_results<BidIt>& operator*();
    const match_results<BidIt> *operator->();
    regex_iterator& operator++();
    regex_iterator& operator++(int);

    BidIt begin;                           // exposition only
    BidIt end;                             // exposition only
    regex_type *pregex;                    // exposition only
    regex_constants::match_flag_type flags; // exposition only
    match_results<BidIt> match;            // exposition only
    };
```

Constructor:

regex_iterator::regex_iterator:

```
regex_iterator();
regex_iterator(BidIt first, BidIt last,
    const regex_type& re,
    regex_constants::match_flag_type f = regex_constants::match_default);
```

The first constructor constructs an end-of-sequence iterator. The second constructor initializes the stored value `begin` with `first`, the stored value `end` with `last`, the stored value `pregex` with `&re`, and the stored value `flags` with `f`. It then calls `regex_search(begin, end, match, *pregex, flags)`. If the search fails, the constructor sets the object to an end-of-sequence iterator.

Types:

regex_iterator::difference_type:

```
typedef std::ptrdiff_t difference_type;
```

The type is a synonym for `std::ptrdiff_t`.

regex_iterator::iterator_category:

```
typedef std::forward_iterator_tag iterator_category;
```

The type is a synonym for `std::forward_iterator_tag`.

regex_iterator::pointer:

```
typedef match_results<BidIt> *pointer;
```

The type is a synonym for `match_results<BidIt>*`, where `BidIt` is the template parameter.

regex_iterator::reference:

```
typedef match_results<BidIt>& reference;
```

The type is a synonym for `match_results<BidIt>&`, where `BidIt` is the template parameter.

regex_iterator::regex_type:

```
typedef basic_regex<Elem, RXtraits> regex_type;
```

The typedef is a synonym for `basic_regex<Elem, RXtraits>`.

regex_iterator::value_type:

```
typedef match_results<BidIt> value_type;
```

The type is a synonym for `match_results<BidIt>`, where `BidIt` is the template parameter.

Member functions:

regex_iterator::operator==:

```
bool operator==(const regex_iterator& right);
```

The member function returns true if `*this` and `right` are both end-of-sequence iterators or if neither is an end-of-sequence iterator and `begin == right.begin`, `end == right.end`, `pregex == right.pregex`, and `flags == right.flags`. Otherwise it returns false.

regex_iterator::operator!=:

```
bool operator!=(const regex_iterator& right);
```

The member function returns `!(*this == right)`.

regex_iterator::operator:*

```
const match_results<BidIt>& operator*();
```

The member function returns the stored value `match`.

regex_iterator::operator->:

```
const match_results<BidIt> *operator->();
```

The member function returns the address of the stored value `match`.

Operators:

regex_iterator::operator++:

```
regex_iterator& operator++();
regex_iterator& operator++(int);
```

If the current match has no characters the first operator calls `regex_search(begin, end, match, *pregex, flags | regex_constants::match_prev_avail | regex_constants::match_not_null)`; otherwise it advances the stored value `begin` to point to the first character after the current match then calls `regex_search(begin, end, match, *pregex, flags | regex_constants::match_prev_avail)`. In either case, if the search fails the operator sets the object to an end-of-sequence iterator. The operator returns the object.

The second operator makes a copy of the object, increments the object, then returns the copy.

regex_token_iterator

Description: The template class describes a constant forward iterator object. Conceptually, it holds a `regex_iterator` object that it uses to search for regular expression matches in a character sequence. It extracts objects of type `sub_match<BidIt>` representing the submatches identified by the index values in the stored vector `subs` for each regular expression match.

An index value of -1 designates the character sequence beginning immediately after the end of the previous regular expression match, or beginning at the start of the character sequence if there was no previous regular expression match, and extending to but not including the first character of the current regular expression match, or to the end of the character sequence if there is no current match. Any other index value `idx` designates the contents of the capture group held in `it.match[idx]`.

Synopsis:

```
template<class BidIt, class Elem = iterator_traits<BidIt>::value_type,
    class RXtraits = regex_traits<Elem> >
        class regex_token_iterator {
public:
    typedef basic_regex<Elem, RXtraits> regex_type;
    typedef sub_match<BidIt> value_type;
    typedef std::forward_iterator_tag iterator_category;
    typedef std::ptrdiff_t difference_type;
```

```
    typedef const sub_match<BidIt> *pointer;
    typedef const sub_match<BidIt>& reference;

    regex_token_iterator();
    regex_token_iterator(BidIt first, BidIt last,
        const regex_type& re, int submatch = 0,
        regex_constants::match_flag_type f = regex_constants::match_default);
    regex_token_iterator(BidIt first, BidIt last,
        const regex_type& re, const std::vector<int> submatches,
        regex_constants::match_flag_type f = regex_constants::match_default);
    template<std::size_t N>
    regex_token_iterator(BidIt first, BidIt last,
        const regex_type& re, const int (&submatches)[N],
        regex_constants::match_flag_type f = regex_constants::match_default);

    bool operator==(const regex_token_iterator& right);
    bool operator!=(const regex_token_iterator& right);

    const basic_string<Elem>& operator*();
    const basic_string<Elem> *operator->();
    regex_token_iterator& operator++();
    regex_token_iterator& operator++(int);

private:
    regex_iterator<BidIt, Elem, RXtraits> it; // exposition only
    vector<int> subs;                          // exposition only
    int pos;                                   // exposition only
    };
```

Constructor:

regex_token_iterator::regex_token_iterator:

```
regex_token_iterator();
regex_token_iterator(BidIt first, BidIt last,
    const regex_type& re, int submatch = 0,
    regex_constants::match_flag_type f = regex_constants::match_default);
regex_token_iterator(BidIt first, BidIt last,
    const regex_type& re, const vector<int> submatches,
    regex_constants::match_flag_type f = regex_constants::match_default);
template<std::size_t N>
regex_token_iterator(BidIt first, BidIt last,
    const regex_type& re, const int (&submatches)[N],
    regex_constants::match_flag_type f = regex_constants::match_default);
```

The first constructor constructs an end-of-sequence iterator.

The second constructor constructs an object whose stored iterator `it` is initialized to `regex_iterator<BidIt, Elem, RXtraits<(first, last, re, f)`, whose stored vector `subs` holds exactly one integer, with value `submatch`, and whose stored value `pos` is 0. Note: the resulting object extracts the submatch identified by the **index value** `submatch` for each successful regular expression match.

The third constructor constructs an object whose stored iterator `it` is initialized to `regex_iterator<BidIt, Elem, RXtraits<(first, last, re, f)`, whose stored vector `subs` holds a copy of the constructor argument `submatches`, and whose stored value `pos` is 0.

The fourth constructor constructs an object whose stored iterator `it` is initialized to `regex_iterator<BidIt, Elem, RXtraits<(first, last, re, f)`, whose stored vector `subs` holds the `N` values pointed to by the constructor argument `submatches`, and whose stored value `pos` is 0.

Types:

regex_token_iterator::difference_type:

```
typedef std::ptrdiff_t difference_type;
```

The type is a synonym for `std::ptrdiff_t`.

regex_token_iterator::iterator_category:

```
typedef std::forward_iterator_tag iterator_category;
```

The type is a synonym for `std::forward_iterator_tag`.

regex_token_iterator::pointer:

```
typedef sub_match<BidIt> *pointer;
```

The type is a synonym for `sub_match<BidIt>*`, where `BidIt` is the template parameter.

regex_token_iterator::reference:

```
typedef sub_match<BidIt>& reference;
```

The type is a synonym for `sub_match<BidIt>&`, where `BidIt` is the template parameter.

regex_token_iterator::regex_type:

```
typedef basic_regex<Elem, RXtraits> regex_type;
```

The typedef is a synonym for `basic_regex<Elem, RXtraits>`.

regex_token_iterator::value_type:

```
typedef sub_match<BidIt> value_type;
```

The type is a synonym for `sub_match<BidIt>`, where `BidIt` is the template parameter.

Member functions:

regex_token_iterator::operator==:

```
bool operator==(const regex_token_iterator& right);
```

The member function returns `it == right.it && subs == right.subs && pos == right.pos`.

regex_token_iterator::operator!=:

```
bool operator!=(const regex_token_iterator& right);
```

The member function returns `!(*this == right)`.

regex_token_iterator::operator:*

```
const sub_match<BidIt>& operator*();
```

The member function returns a `sub_match<BidIt>` object representing the capture group identified by the index value `subs[pos]`.

regex_token_iterator::operator->:

```
const sub_match<BidIt> *operator->();
```

The member function returns a pointer to a `sub_match<BidIt>` object representing the capture group identified by the index value `subs[pos]`.

Operators:

regex_token_iterator::operator++:

```
regex_token_iterator& operator++();
regex_token_iterator& operator++(int);
```

If the stored iterator `it` is an end-of-sequence iterator the first operator sets the stored value `pos` to the value of `subs.size()` (thus making an end-of-sequence iterator). Otherwise the operator increments the stored value `pos`; if the result is equal to the value `subs.size()` it sets the stored value `pos` to 0 and increments the stored iterator `it`. If incrementing the stored iterator leaves it unequal to an end-of-sequence iterator the operator does nothing further. Otherwise, if the end of the preceding match was at the end of the character sequence the operator sets the stored value of `pos` to `subs.size()`. Otherwise, the operator repeatedly increments the stored value `pos` until `pos == subs.size()` or `subs[pos] == -1` (thus ensuring that the next dereference of the iterator will return the tail of the character sequence if one of the index values is -1). In all cases the operator returns the object.

The second operator makes a copy of the object, increments the object, then returns the copy.

regex_traits

Description: The template class describes various **regular expression traits** for type *Elem*. The template class "basic_regex" on page 300 uses this information to manipulate elements of type *Elem*.

Each `regex_traits` object holds an object of type `regex_traits::locale` which is used by some of its member functions. The **default locale** is a copy of `regex_traits::locale()`. The member function `imbue` replaces the locale object, and the member function `getloc` returns a copy of the locale object.

Synopsis:

```
template<class Elem>
    struct regex_traits {
    regex_traits();

    static size_type length(const char_type *str);
    char_type translate(char_type ch) const;
    char_type translate_nocase(char_type ch) const;
    template<class FwdIt>
        string_type transform(FwdIt first, FwdIt last) const;
    template<class FwdIt>
        string_type transform_primary(FwdIt first, FwdIt last) const;

    template<class FwdIt>
        char_class_type lookup_classname(FwdIt first, FwdIt last) const;
    template<class FwdIt>
        string_type lookup_collatename(FwdIt first, FwdIt last) const;
    bool isctype(char_type ch, char_class_type cls) const;

    int value(Elem ch, int base) const;

    locale_type imbue(locale_type loc);
```

```
locale_type getloc() const;

typedef Elem char_type;
typedef T6 size_type;
typedef basic_string<Elem> string_type;
typedef T7 locale_type;
typedef T8 char_class_type;
};
```

Constructor:

regex_traits::regex_traits:

```
regex_traits();
```

The constructor constructs an object whose stored `locale` object is initialized to the default locale.

Types:

regex_traits::char_class_type:

```
typedef T8 char_class_type;
```

The type is a synonym for an unspecified type that designates character classes. Values of this type can be combined using the **|** operator to designate character classes that are the union of the classes designated by the operands.

regex_traits::char_type:

```
typedef Elem char_type;
```

The typedef is a synonym for the template argument `Elem`.

regex_traits::locale_type:

```
typedef T7 locale_type;
```

The typedef is a synonym for a type that encapsulates locales. In the specializations `regex_traits<char>` and `regex_traits<wchar_t>` it is a synonym for `std::locale`.

regex_traits::size_type:

```
typedef T6 size_type;
```

The typedef is a synonym for an unsigned integral type. In the specializations `regex_traits<char>` and `regex_traits<wchar_t>` it is a synonym for `std::size_t`.

The typedef is a synonym for `std::size_t`.

regex_traits::string_type:

```
typedef basic_string<Elem> string_type;
```

The typedef is a synonym for `basic_string<Elem>`.

Member functions:

regex_traits::getloc:

```
locale_type getloc() const;
```

The member function returns the stored `locale` object.

regex_traits::imbue:

```
locale_type imbue(locale_type loc);
```

The member function copies `loc` to the stored `locale` object and returns a copy of the previous value of the stored `locale` object.

regex_traits::isctype:

```
bool isctype(char_type ch, char_class_type cls) const;
```

The member function returns true only if the character `ch` is in the character class designated by `cls`.

regex_traits::length:

```
static size_type length(const char_type *str);
```

The static member function returns `std::char_traits<char_type>::length(str)`.

regex_traits::lookup_classname:

```
template<class FwdIt>
    char_class_type lookup_classname(FwdIt first, FwdIt last) const;
```

The member function returns a value that designates the character class named by the character sequence pointed to by its arguments. The value does not depend on the case of the characters in the sequence.

The specialization `regex_traits<char>` recognizes the names `"d"`, `"s"`, `"w"`, `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"`, and `"xdigit"`, all without regard to case.

The specialization `regex_traits<wchar_t>` recognizes the names `L"d"`, `L"s"`, `L"w"`, `L"alnum"`, `L"alpha"`, `L"blank"`, `L"cntrl"`, `L"digit"`, `L"graph"`, `L"lower"`, `L"print"`, `L"punct"`, `L"space"`, `L"upper"`, and `L"xdigit"`, all without regard to case.

regex_traits::lookup_collatename:

```
template<class FwdIt>
string_type lookup_collatename(FwdIt first, FwdIt last) const;
```

xxx

regex_traits::value:

```
int value(Elem ch, int radix) const;
```

The member function returns the value represented by the character `ch` in the base `radix`, or -1 if `ch` is not a valid digit in the base `radix`. The function will only be called with a `radix` argument of 8, 10, or 16.

regex_traits::transform:

```
template<class FwdIt>
string_type transform(FwdIt first, FwdIt last) const;
```

The member function returns a string that it generates by using a transformation rule that depends on the stored `locale` object. For two character sequences designated by the iterator ranges `[first1, last1)` and `[first2, last2)`, `transform(first1, last1) < transform(first2, last2)` if the character sequence designated by the iterator range `[first1, last1)` sorts before the character sequence designated by the iterator range `[first2, last2)`.

regex_traits::transform_primary:

```
template<class FwdIt>
string_type transform_primary(FwdIt first, FwdIt last) const;
```

The member function returns a string that it generates by using a transformation rule that depends on the stored `locale` object. For two character sequences designated by the iterator ranges `[first1, last1)` and `[first2, last2)`, `transform_primary(first1, last1) < transform_primary(first2, last2)` if the character sequence designated by the iterator range `[first1, last1)` sorts before the character sequence designated by the iterator range `[first2, last2)` without regard for case or accents.

regex_traits::translate:

```
char_type translate(char_type ch) const;
```

The member function returns a character that it generates by using a transformation rule that depends on the stored `locale` object. For two `char_type` objects `ch1` and `ch2`, `translate(ch1) == translate(ch2)` only if `ch1` and `ch2` should match when one occurs in the regular expression definition and the other occurs at a corresponding position in the target sequence for a (locale-sensitive) match.

regex_traits::translate_nocase:

```
char_type translate_nocase(char_type ch) const;
```

The member function returns a character that it generates by using a transformation rule that depends on the stored `locale` object. For two `char_type` objects `ch1` and `ch2`, `translate_nocase(ch1) == translate_nocase(ch2)` only if `ch1` and `ch2` should match when one occurs in the regular expression definition and the other occurs at a corresponding position in the target sequence for a (case-insensitive) match.

regex_traits<char>

```
template <>
    class regex_traits<char>
```

The class is an explicit specialization of template class “regex_traits” on page 314 for elements of type *char* (so that it can take advantage of library functions that manipulate objects of this type).

regex_traits<wchar_t>

```
template <>
    class regex_traits<wchar_t>
```

The class is an explicit specialization of template class “regex_traits” on page 314 for elements of type *wchar_t* (so that it can take advantage of library functions that manipulate objects of this type).

sub_match

Description: The template class describes an object that designates a sequence of characters that matched a capture group in a call to “regex_match” on page 319 or to “regex_search” on page 320. Objects of type “match_results” on page 304 hold an array of these objects, one for each capture group in the regular expression that was used in the search.

If the capture group was not matched the object's data member `matched` holds false, and the two iterators `first` and `second` (inherited from the base `std::pair`) are equal. If the capture group was matched, `matched` holds true, the iterator `first` points to the first character in the target sequence that matched the capture group, and the iterator `second` points one position past the last character in the target sequence that matched the capture group. Note that for a zero-length match the member `matched` holds true, the two iterators will be equal, and both will point to the position of the match.

A **zero-length** match can occur when a capture group consists solely of an assertion, or of a repetition that allows zero repeats. For example:

- **"^"** matches the target sequence "a"; the `sub_match` object corresponding to capture group 0 holds iterators that both point to the first character in the sequence.
- **"b(a*)b"** matches the target sequence "bb"; the `sub_match` object corresponding to capture group 1 holds iterators that both point to the second character in the sequence.

Synopsis:

```
template<class BidIt>
    class sub_match : public std::pair<BidIt, BidIt>{
public:
    bool matched;

    int compare(const sub_match& right) const;
    int compare(const basic_string<value_type>& right) const;
    int compare(const value_type *right) const;

    difference_type length() const;
    operator basic_string<value_type>() const;
    basic_string<value_type> str() const;

    typedef typename iterator_traits<BidIt>::value_type value_type;
    typedef typename iterator_traits<BidIt>::difference_type difference_type;
    typedef BidIt iterator;
    };
```

Types:

sub_match::difference_type:

```
typedef typename iterator_traits<BidIt>::difference_type difference_type;
```

The typedef is a synonym for `iterator_traits<BidIt>::difference_type`.

sub_match::iterator:

```
typedef BidIt iterator;
```

The typedef is a synonym for the template type argument `Bidit`.

sub_match::value_type:

```
typedef typename iterator_traits<BidIt>::value_type value_type;
```

The typedef is a synonym for `iterator_traits<BidIt>::value_type`.

Member functions:

sub_match::compare:

```
int compare(const sub_match& right) const;
int compare(const basic_string<value_type>& right) const;
int compare(const value_type *right) const;
```

The first member function compares the matched sequence [first, second) to the matched sequence [right.first, right.second). The second member function compares the matched sequence [first, second) to the character sequence [right.begin(), right.end()). The third member function compares the matched sequence [first, second) to the character sequence [right, right + std::char_traits<value_type>::length(right)).

Each function returns:
- a negative value if the first differing value in the matched sequence compares less than the corresponding element in the operand sequence (as determined by std::char_traits<value_type>::compare), or if the two have a common prefix but the target sequence is longer
- zero if the two compare equal element by element and have the same length
- a positive value otherwise

sub_match::length:

```
difference_type length() const;
```

The member function returns the length of the matched sequence, or 0 if there was no matched sequence.

sub_match::str:

```
basic_string<value_type> str() const;
```

The member function returns basic_string<value_type>(first, second).

Members:

sub_match::matched:

```
bool matched;
```

The member holds true only if the capture group associated with *this was part of the regular expression match.

Operators:

sub_match::operator basic_string<value_type>:

```
operator basic_string<value_type>() const;
```

The member operator returns str().

Template functions

regex_match

```
template<class BidIt, class Alloc, class Elem, class RXtraits, class Alloc2>
    bool regex_match(BidIt first, Bidit last,
        match_results<BidIt, Alloc>& match,
        const basic_regex<Elem, RXtraits, Alloc2>& re,
        match_flag_type flags = match_default);
template<class BidIt, class Elem, class RXtraits, class Alloc2>
    bool regex_match(BidIt first, Bidit last,
        const basic_regex<Elem, RXtraits, Alloc2>& re,
        match_flag_type flags = match_default);
template<class Elem, class Alloc, class RXtraits, class Alloc2>
    bool regex_match(const Elem *ptr,
        match_results<const Elem*, Alloc>& match,
        const basic_regex<Elem, RXtraits, Alloc2>& re,
```

```
        match_flag_type flags = match_default);
template<class Elem, class RXtraits, class Alloc2>
    bool regex_match(const Elem *ptr,
        const basic_regex<Elem, RXtraits, Alloc2>& re,
        match_flag_type flags = match_default);
template<class IOtraits, class IOalloc, class Alloc, class Elem,
    class RXtraits, class Alloc2>
    bool regex_match(
    const basic_string<Elem, IOtraits, IOalloc>& str,
    match_results<typename basic_string<Elem, IOtraits, IOalloc>::const_iterator,
    Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);
template<class IOtraits, class IOalloc, class Elem, class RXtraits, class Alloc2>
    bool regex_match(const basic_string<Elem, IOtraits, IOalloc>& str,
        const basic_regex<Elem, RXtraits, Alloc2>& re,
        match_flag_type flags = match_default);
```

Each template function returns true only if its operand sequence exactly matches its regular expression argument `re`. The functions that take a `match_results` object set its members to reflect whether the match succeeded and if so what the various capture groups in the regular expression captured.

regex_replace

```
template<class OutIt, class BidIt, class RXtraits, class Alloc, class Elem>
    OutIt regex_replace(OutIt out,
        BidIt first, BidIt last,
        const basic_regex<Elem, RXtraits, Alloc>& re,
        const basic_string<Elem>& fmt,
        match_flag_type flags = match_default);
template<class RXtraits, class Alloc, class Elem>
    basic_string<Elem> regex_replace(const basic_string<Elem>& str,
        const basic_regex<Elem, RXtraits, Alloc>& re,
        const basic_string<Elem>& fmt,
        match_flag_type flags = match_default);
```

The first function constructs a "regex_iterator" on page 309 object `iter(first, last, re, flags)` and uses it to split its input range `[first, last)` into a series of subsequences $T_0M_0T_1M_1...T_{N-1}M_{N-1}T_N$, where M_n is the n^{th} match detected by the iterator. If no matches are found, T_0 is the entire input range and `N` is 0. If `(flags & format_first_only) != 0` only the first match is used, T_1 is all of the input text that follows the match, and `N` is 1. For each `i` in the range `[0, N)`, if `(flags & format_no_copy) == 0` it copies the text in the range T_i to the iterator `out`. It then calls `m.format(out, fmt, flags)`, where `m` is the `match_results` object returned by the iterator object `iter` for the subsequence M_i. Finally, if `(flags & format_no_copy) == 0` it copies the text in the range T_N to the iterator `out`. The function returns `out`.

The second function constructs a local variable `result` of type `basic_string<charT>` and calls `regex_replace(back_inserter(result), str.begin(), str.end(), re, fmt, flags)`. It returns `result`.

regex_search

```
template<class BidIt, class Alloc, class Elem, class RXtraits>
    bool regex_search(BidIt first, Bidit last,
        match_results<BidIt, Alloc>& match,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);
 template<class BidIt, class Elem, class RXtraits>
    bool regex_search(BidIt first, Bidit last,
        const basic_regex<Elem, RXtraits>& re,
        match_flag_type flags = match_default);
 template<class Elem, class Alloc, class RXtraits>
```

```
   bool regex_search(const Elem* ptr,
       match_results<const Elem*, Alloc>& match,
       const basic_regex<Elem, RXtraits>& re,
       match_flag_type flags = match_default);
template<class Elem, class RXtraits>
   bool regex_search(const Elem* ptr,
       const basic_regex<Elem, RXtraits>& re,
       match_flag_type flags = match_default);
template<class IOtraits, class IOalloc, class Alloc, class Elem, class RXtraits>
   bool regex_search(
       const basic_string<Elem, IOtraits, IOalloc>& str,
       match_results<typename basic_string<Elem, IOtraits, IOalloc>::
       const_iterator, Alloc>& match, const basic_regex<Elem, RXtraits>& re,
       match_flag_type flags = match_default);
template<class IOtraits, class IOalloc, class Elem, class RXtraits
>    bool regex_search(
       const basic_string<Elem, IOtraits, IOalloc>& str,
       const basic_regex<Elem, RXtraits>& re,
       match_flag_type flags = match_default);
```

Each template function returns true only if a search for its regular expression argument `re` in its operand sequence succeeds. The functions that take a `match_results` object set its members to reflect whether the search succeeded and if so what the various capture groups in the regular expression captured.

swap

```
template<class Elem, class RXtraits>
   void swap(basic_regex<Elem, RXtraits, Alloc>& left,
       basic_regex<Elem, RXtraits>& right) throw();
template<class Elem, class IOtraits, class BidIt, class Alloc>
   void swap(match_results<BidIt, Alloc>& left,
       match_results<BidIt, Alloc>& right) throw();
```

The template functions swap the contents of their respective arguments in constant time and do not throw exceptions.

Types

cmatch

```
typedef match_results<const char*> cmatch;
```

The type describes a specialization of template class "match_results" on page 304 for iterators of type *const char**.

cregex_iterator

```
typedef regex_iterator<const char*> cregex_iterator;
```

The type describes a specialization of template class "regex_iterator" on page 309 for iterators of type *const char**.

cregex_token_iterator

```
typedef regex_token_iterator<const char*> cregex_token_iterator;
```

The type describes a specialization of template class "regex_token_iterator" on page 311 for iterators of type *const char**.

csub_match

```
typedef sub_match<const char*> csub_match;
```

The type describes a specialization of template class "sub_match" on page 317 for iterators of type *const char**.

regex

```
typedef basic_regex<char> regex;
```

The type describes a specialization of template class "basic_regex" on page 300 for elements of type *char*.

regex_constants

```
namespace regex_constants {
  typedef T1 syntax_option_type;
  typedef T2 match_flag_type;
  typedef T3 error_type;
  }
```

regex_constants::error_type:

```
typedef T3 error_type;
static const error_type error_badbrace, error_badrepeat, error_brace,
  error_brack, error_collate, error_complexity, error_ctype,
  error_escape, error_paren, error_range, error_space,
  error_stack, error_backref;
```

The type is an enumerated type that describes an object that can hold **error flags**. The distinct flag values are:

- **error_badbrace —** the expression contained an invalid count in a { } expression
- **error_badrepeat —** a repeat expression (one of '*', '?', '+', '{' in most contexts) was not preceded by an expression
- **error_brace —** the expression contained an unmatched '{' or '}'
- **error_brack —** the expression contained an unmatched '[' or ']'
- **error_collate —** the expression contained an invalid collating element name
- **error_complexity —** an attempted match failed because it was too complex
- **error_ctype —** the expression contained an invalid character class name
- **error_escape —** the expression contained an invalid escape sequence
- **error_paren —** the expression contained an unmatched '(' or ')'
- **error_range —** the expression contained an invalid character range specifier
- **error_space —** parsing a regular expression failed because there were not enough resources available
- **error_stack —** an attempted match failed because there was not enough memory available
- **error_backref —** the expression contained an invalid back reference

regex_constants::match_flag_type:

```
typedef T2 match_flag_type;
static const match_flag_type match_any, match_default, match_not_bol,
  match_not_bow, match_continuous, match_not_eol, match_not_eow,
  match_not_null, match_partial, match_prev_avail;
```

The type is a bitmask type that describes options to be used when matching a text sequence against a regular expression and format flags to be used when replacing text. Options can be combined with `|`.

The match options are:

- **match_default**

- **match_not_bol —** do not treat the first position in the target sequence as the beginning of a line
- **match_not_eol —** do not treat the past-the-end position in the target sequence as the end of a line
- **match_not_bow —** do not treat the first position in the target sequence as the beginning of a word
- **match_not_eow —** do not treat the past-the-end position in the target sequence as the end of a word
- **match_any —** if more than one match is possible any match is acceptable
- **match_not_null —** do not treat an empty subsequence as a match
- **match_continuous —** do not search for matches other than at the beginning of the target sequence
- **match_prev_avail —** `--first` is a valid iterator; ignore `match_not_bol` and `match_not_bow` if set

The **format flags** are:

- **format_default —** use ECMAScript format rules
- **format_sed —** use sed format rules
- **format_no_copy —** do not copy text that does not match the regular expression
- **format_first_only —** do not search for matches after the first one

regex_constants::syntax_option_type:

```
typedef T1 syntax_option_type;
static const syntax_option_type awk, basic, collate, ECMAScript,
  egrep, extended, grep, icase, nosubs, optimize;
```

The type is a bitmask type that describes language specifiers and syntax modifiers to be used when compiling a regular expression. Options can be combined with `|`. No more than one language specifier should be used at a time.

The language specifiers are:

- **basic —** compile as BRE
- **extended —** compile as ERE
- **ECMAScript —** compile as ECMAScript
- **awk —** compile as awk
- **grep —** compile as grep
- **egrep —** compile as egrep

The syntax modifiers are:

- **icase —** make matches case-insensitive
- **nosubs —** the implementaton need not keep track of the contents of capture groups
- **optimize —** the implementation should emphasize speed of matching rather than speed of regular expression compilation
- **collate —** make matches locale-sensitive

smatch

```
typedef match_results<string::const_iterator> smatch;
```

The type describes a specialization of template class “match_results” on page 304 for iterators of type *string::const_iterator*.

sregex_iterator

```
typedef regex_iterator<string::const_iterator> sregex_iterator;
```

The type describes a specialization of template class "regex_iterator" on page 309 for iterators of type *string::const_iterator*.

sregex_token_iterator

```
typedef regex_token_iterator<string::const_iterator> sregex_token_iterator;
```

The type describes a specialization of template class "regex_token_iterator" on page 311 for iterators of type *string::const_iterator*.

ssub_match

```
typedef sub_match<string::const_iterator> ssub_match;
```

The type describes a specialization of template class "sub_match" on page 317 for iterators of type *string::const_iterator*.

wcmatch

```
typedef match_results<const wchar_t *> wcmatch;
```

The type describes a specialization of template class "match_results" on page 304 for iterators of type *const wchar_t**.

wcregex_iterator

```
typedef regex_iterator<const wchar_t*> wcregex_iterator;
```

The type describes a specialization of template class "regex_iterator" on page 309 for iterators of type *const wchar_t**.

wcregex_token_iterator

```
typedef regex_token_iterator<const wchar_t*> wcregex_token_iterator;
```

The type describes a specialization of template class "regex_token_iterator" on page 311 for iterators of type *const wchar_t**.

wcsub_match

```
typedef sub_match<const wchar_t*> wcsub_match;
```

The type describes a specialization of template class "sub_match" on page 317 for iterators of type *const wchar_t**.

wsmatch

```
typedef match_results<wstring::const_iterator> wsmatch;
```

The type describes a specialization of template class "match_results" on page 304 for iterators of type *wstring::const_iterator*.

wsregex_iterator

```
typedef regex_iterator<wstring::const_iterator> wsregex_iterator;
```

The type describes a specialization of template class "regex_iterator" on page 309 for iterators of type *wstring::const_iterator*.

wsregex_token_iterator

```
typedef regex_token_iterator<wstring::const_iterator> wsregex_token_iterator;
```

The type describes a specialization of template class "regex_token_iterator" on page 311 for iterators of type *wstring::const_iterator*.

wssub_match

```
typedef sub_match<wstring::const_iterator> wssub_match;
```

The type describes a specialization of template class "sub_match" on page 317 for iterators of type *wstring::const_iterator*.

wregex

```
typedef basic_regex<wchar_t> wregex;
```

The type describes a specialization of template class "basic_regex" on page 300 for elements of type *wchar_t*.

Operators

operator==

```
template<class BidIt>
    bool operator==(
        const sub_match<BidIt>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator==(
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator==(
        const sub_match<BidIt>& left,
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& right);
template<class BidIt>
    bool operator==(
        const typename iterator_traits<BidIt>::value_type* left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator==(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type* right);
template<class BidIt>
    bool operator==(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator==(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type& right);
template<class BidIt, class Alloc>
    bool operator==(
        const match_results<BidIt, Alloc>& left,
        const match_results<BidIt, Alloc>& right);
```

Each template operator converts each of its arguments to a string type and returns the result of comparing the converted objects for equality.

When a template operator **converts** its arguments to a string type it uses the first of the following transformations that applies:

- arguments whose types are a specialization of template class `sub_match` are converted by calling the `str` member function;

- arguments whose types are a specialization of the template class `basic_string` are unchanged;
- all other argument types are converted by passing the argument value to the constructor for an appropriate specialization of the template class `basic_string`.

operator!=

```
template<class BidIt>
    bool operator!=(
        const sub_match<BidIt>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator!=(
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator!=(
        const sub_match<BidIt>& left,
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& right);
template<class BidIt>
    bool operator!=(
        const typename iterator_traits<BidIt>::value_type *left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator!=(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type *right);
template<class BidIt>
    bool operator!=(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator!=(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type& right);
template<class BidIt, class Alloc>
    bool operator!=(
        const match_results<BidIt, Alloc>& left,
        const match_results<BidIt, Alloc>& right);
```

Each template operator returns `!(left == right)`.

operator<

```
template<class BidIt>
    bool operator<(
        const sub_match<BidIt>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator<(
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator<(
        const sub_match<BidIt>& left,
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& right);
template<class BidIt>
    bool operator<(
        const typename iterator_traits<BidIt>::value_type *left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator<(
        const sub_match<BidIt>& left,
```

```
        const typename iterator_traits<BidIt>::value_type *right);
template<class BidIt>
    bool operator<(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator<(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type& right);
```

Each template operator converts its arguments to a string type and returns true only if the converted value of `left` compares less than the converted value of `right`.

operator<=

```
template<class BidIt>
    bool operator<=(
        const sub_match<BidIt>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator<=(
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator<=(
        const sub_match<BidIt>& left,
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& right);
template<class BidIt>
    bool operator<=(
        const typename iterator_traits<BidIt>::value_type *left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator<=(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type *right);
template<class BidIt>
    bool operator<=(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator<=(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type& right);
```

Each template operator returns `!(right < left)`.

operator>

```
template<class BidIt>
    bool operator>(
        const sub_match<BidIt>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator>(
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator>(
        const sub_match<BidIt>& left,
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& right);
template<class BidIt>
    bool operator>(
```

```
        const typename iterator_traits<BidIt>::value_type *left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator>(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type *right);
template<class BidIt>
    bool operator>(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator>(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type& right);
```

Each template operator returns `right < left`.

operator>=

```
template<class BidIt>
    bool operator>=(
        const sub_match<BidIt>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator>=(
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& left,
        const sub_match<BidIt>& right);
template<class BidIt, class IOtraits, class Alloc>
    bool operator>=(
        const sub_match<BidIt>& left,
        const basic_string<typename iterator_traits<BidIt>::value_type,
        IOtraits, Alloc>& right);
template<class BidIt>
    bool operator>=(
        const typename iterator_traits<BidIt>::value_type *left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator>=(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type *right);
template<class BidIt>
    bool operator>=(
        const typename iterator_traits<BidIt>::value_type& left,
        const sub_match<BidIt>& right);
template<class BidIt>
    bool operator>=(
        const sub_match<BidIt>& left,
        const typename iterator_traits<BidIt>::value_type& right);
```

Each template operator returns `!(left < right)`.

operator<<

```
template<class Elem, class IOtraits, class Alloc, class BidIt>
    basic_ostream<Elem, IOtraits>&
    operator<<(
        basic_ostream<Elem, IOtraits>& os,
        const sub_match<BidIt>& right);
```

The template operator returns `os << right.str()`.

<set>

Description

Include the STL standard header **<set>** to define the container template classes `set` and `multiset`, and their supporting templates.

Synopsis

```
namespace std {
template<class Key, class Pred, class A>
    class set;
template<class Key, class Pred, class A>
    class multiset;

        // TEMPLATE FUNCTIONS
template<class Key, class Pred, class A>
    bool operator==(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator==(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator!=(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator!=(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<=(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<=(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>=(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>=(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    void swap(
```

```
        set<Key, Pred, A>& lhs,
        set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    void swap(
        multiset<Key, Pred, A>& lhs,
        multiset<Key, Pred, A>& rhs);
    }
```

Macros

__IBM_FAST_SET_MAP_ITERATOR

```
#define __IBM_FAST_SET_MAP_ITERATOR
```

The __IBM_FAST_SET_MAP_ITERATOR macro enables doubly-linked list data structures for the Standard C++ Library set and map classes and can assist with improving performance. By default, this macro must be explicitly specified to gain any potential performance improvement. Any code compiled with this macro cannot be mixed with code compiled without the macro.

Classes

multiset

Description: The template class describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is ordered by the predicate `Pred`. Each element serves as both a **sort key** and a **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this stored object by calling the member function `key_comp()`. Such a function object must impose a total ordering on sort keys of type `Key`. For any element `x` that precedes `y` in the sequence, `key_comp()(y, x)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class set, an object of template class `multiset` does not ensure that `key_comp()(x, y)` is true. (Keys need not be unique.)

The object allocates and frees storage for the sequence it controls through a stored allocator object of class `A`. Such an allocator object must have the same external interface as an object of template class allocator. Note that the stored allocator object is *not* copied when the container object is assigned.

Synopsis:

```
template<class Key, class Pred = less<Key>,
    class A = allocator<Key> >
    class multiset {
public:
    typedef Key key_type;
    typedef Pred key_compare;
    typedef Key value_type;
    typedef Pred value_compare;
    typedef A allocator_type;
    typedef A::pointer pointer;
    typedef A::const_pointer const_pointer;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
```

```
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
    multiset();
    explicit multiset(const Pred& comp);
    multiset(const Pred& comp, const A& al);
    multiset(const multiset& x);
    template<class InIt>
        multiset(InIt first, InIt last);
    template<class InIt>
        multiset(InIt first, InIt last,
            const Pred& comp);
    template<class InIt>
        multiset(InIt first, InIt last,
            const Pred& comp, const A& al);
    const_iterator begin() const;
    const_iterator end() const;
    const_reverse_iterator rbegin() const;
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    iterator insert(const value_type& x);
    iterator insert(iterator it, const value_type& x);
    template<class InIt>
        void insert(InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& key);
    void clear();
    void swap(multiset& x);
    key_compare key_comp() const;
    value_compare value_comp() const;
    const_iterator find(const Key& key) const;
    size_type count(const Key& key) const;
    const_iterator lower_bound(const Key& key) const;
    const_iterator upper_bound(const Key& key) const;
    pair<const_iterator, const_iterator>
        equal_range(const Key& key) const;
    };
```

Constructor:

multiset::multiset:

```
multiset();
explicit multiset(const Pred& comp);
multiset(const Pred& comp, const A& al);
multiset(const multiset& x);
template<class InIt>
    multiset(InIt first, InIt last);
template<class InIt>
    multiset(InIt first, InIt last,
        const Pred& comp);
template<class InIt>
    multiset(InIt first, InIt last,
        const Pred& comp, const A& al);
```

All constructors store an allocator object and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

All constructors also store a function object that can later be returned by calling `key_comp()`. The function object is the argument `comp`, if present. For the copy constructor, it is `x.key_comp()`). Otherwise, it is `Pred()`.

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by `x`. The last three constructors specify the sequence of element values `[first, last)`.

Types:

multiset::allocator_type:

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

multiset::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

multiset::const_pointer:

```
typedef A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

multiset::const_reference:

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

multiset::const_reverse_iterator:

```
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

multiset::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

multiset::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

multiset::key_compare:

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of two elements in the controlled sequence.

multiset::key_type:

```
typedef Key key_type;
```

The type describes the sort key object which constitutes each element of the controlled sequence.

multiset::pointer:

```
typedef A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

multiset::reference:

```
typedef A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

multiset::reverse_iterator:

```
typedef reverse_iterator<iterator> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

multiset::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

multiset::value_compare:

```
typedef Pred value_compare;
```

The type describes a function object that can compare two elements as sort keys to determine their relative order in the controlled sequence.

multiset::value_type:

```
typedef Key value_type;
```

The type describes an element of the controlled sequence.

Member functions:

multiset::begin:

```
const_iterator begin() const;
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

multiset::clear:

```
void clear();
```

The member function calls `erase( begin(), iset::end())`.

multiset::count:

```
size_type count(const Key& key) const;
```

The member function returns the number of elements `x` in the range `[lower_bound(key), upper_bound(key))`.

multiset::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

multiset::end:

```
const_iterator end() const;
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

multiset::equal_range:

```
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
```

The member function returns a pair of iterators `x` such that `x.first == lower_bound(key)` and `x.second == upper_bound(key)`.

multiset::erase:

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes the elements with sort keys in the range `[lower_bound(key), upper_bound(key))`. It returns the number of elements it removes.

The member functions never throw an exception.

multiset::find:

```
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering to `key`. If no such element exists, the function returns `end()`.

multiset::get_allocator:

```
A get_allocator() const;
```

The member function returns the stored allocator object.

multiset::insert:

```
iterator insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
```

The first member function inserts the element `x` in the controlled sequence, then returns the iterator that designates the inserted element. The second member function returns `insert(x)`, using `it` as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows `it`.) The third member function inserts the sequence of element values, for each `it` in the range `[first, last)`, by calling `insert(*it)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

multiset::key_comp:

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object defines the member function:

```
bool operator(const Key& x, const Key& y);
```

which returns true if `x` strictly precedes `y` in the sort order.

multiset::lower_bound:

```
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(x, key)` is false.

If no such element exists, the function returns `end()`.

multiset::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

multiset::rbegin:

```
const_reverse_iterator rbegin() const;
```

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

multiset::rend:

```
const_reverse_iterator rend() const;
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

multiset::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

multiset::swap:

```
void swap(multiset& x);
```

The member function swaps the controlled sequences between `*this` and `x`. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type `Pred`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

multiset::upper_bound:

```
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(key, x)` is true.

If no such element exists, the function returns `end()`.

multiset::value_comp:

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

set

Description: The template class describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is ordered by the predicate `Pred`. Each element serves as both a **sort key** and a **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this stored object by calling the member function `key_comp()`. Such a function object must impose a total ordering on sort keys of type `Key`. For any element `x` that precedes `y` in the sequence, `key_comp()(y, x)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class multiset, an object of template class `set` ensures that `key_comp()(x, y)` is true. (Each key is unique.)

The object allocates and frees storage for the sequence it controls through a stored allocator object of class `A`. Such an allocator object must have the same external interface as an object of template class allocator. Note that the stored allocator object is *not* copied when the container object is assigned.

Synopsis:

```
template<class Key, class Pred = less<Key>,
    class A = allocator<Key> >
    class set {
public:
    typedef Key key_type;
    typedef Pred key_compare;
    typedef Key value_type;
    typedef Pred value_compare;
    typedef A allocator_type;
    typedef A::pointer pointer;
    typedef A::const_pointer const_pointer;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
    set();
    explicit set(const Pred& comp);
    set(const Pred& comp, const A& al);
    set(const set& x);
    template<class InIt>
        set(InIt first, InIt last);
    template<class InIt>
        set(InIt first, InIt last,
            const Pred& comp);
    template<class InIt>
        set(InIt first, InIt last,
            const Pred& comp, const A& al);
    const_iterator begin() const;
    const_iterator end() const;
    const_reverse_iterator rbegin() const;
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    pair<iterator, bool> insert(const value_type& x);
    iterator insert(iterator it, const value_type& x);
    template<class InIt>
        void insert(InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& key);
    void clear();
    void swap(set& x);
    key_compare key_comp() const;
    value_compare value_comp() const;
    const_iterator find(const Key& key) const;
    size_type count(const Key& key) const;
    const_iterator lower_bound(const Key& key) const;
    const_iterator upper_bound(const Key& key) const;
    pair<const_iterator, const_iterator>
        equal_range(const Key& key) const;
    };
```

Constructor:

set::set:

```
set();
explicit set(const Pred& comp);
set(const Pred& comp, const A& al);
set(const set& x);
```

```
template<class InIt>
    set(InIt first, InIt last);
template<class InIt>
    set(InIt first, InIt last,
        const Pred& comp);
template<class InIt>
    set(InIt first, InIt last,
        const Pred& comp, const A& al);
```

All constructors store an allocator object and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

All constructors also store a function object that can later be returned by calling `key_comp()`. The function object is the argument `comp`, if present. For the copy constructor, it is `x.key_comp()`). Otherwise, it is `Pred()`.

The first three constructors specify an empty initial controlled sequence. The fourth constructor specifies a copy of the sequence controlled by `x`. The last three constructors specify the sequence of element values `[first, last)`.

Types:

set::allocator_type:

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

set::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is desciibed here as a synonym for the implementation-defined type `T1`.

set::const_pointer:

```
typedef A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

set::const_reference:

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

set::const_reverse_iterator:

```
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

set::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is desciibed here as a synonym for the implementation-defined type T3.

set::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T0.

set::key_compare:

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of two elements in the controlled sequence.

set::key_type:

```
typedef Key key_type;
```

The type describes the sort key object which constitutes each element of the controlled sequence.

set::pointer:

```
typedef A::const_pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

set::reference:

```
typedef A::const_reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

set::reverse_iterator:

```
typedef reverse_iterator<iterator> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

set::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is desciibed here as a synonym for the implementation-defined type T2.

set::value_compare:

```
typedef Pred value_compare;
```

The type describes a function object that can compare two elements as sort keys to determine their relative order in the controlled sequence.

set::value_type:

```
typedef Key value_type;
```

The type describes an element of the controlled sequence.

Member functions:

set::begin:

```
const_iterator begin() const;
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

set::clear:

```
void clear();
```

The member function calls `erase( begin(), end())`.

set::count:

```
size_type count(const Key& key) const;
```

The member function returns the number of elements `x` in the range `[lower_bound(key), upper_bound(key))`.

set::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

set::end:

```
const_iterator end() const;
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

set::equal_range:

```
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
```

The member function returns a pair of iterators `x` such that `x.first == lower_bound(key)` and `x.second == upper_bound(key)`.

set::erase:

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes the elements with sort keys in the range `[lower_bound(key), upper_bound(key))`. It returns the number of elements it removes.

The member functions never throw an exception.

set::find:

```
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key has equivalent ordering to key. If no such element exists, the function returns end().

set::get_allocator:

```
A get_allocator() const;
```

The member function returns the stored allocator object.

set::insert:

```
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
```

The first member function determines whether an element y exists in the sequence whose key has equivalent ordering to that of x. If not, it creates such an element y and initializes it with x. The function then determines the iterator it that designates y. If an insertion occurred, the function returns pair(it, true). Otherwise, it returns pair(it, false).

The second member function returns insert(x), using it as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows it.) The third member function inserts the sequence of element values, for each it in the range [first, last), by calling insert(*it).

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

set::key_comp:

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object defines the member function:

```
bool operator(const Key& x, const Key& y);
```

which returns true if x strictly precedes y in the sort order.

set::lower_bound:

```
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element x in the controlled sequence for which key_comp()(x, key) is false.

If no such element exists, the function returns end().

set::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

set::rbegin:

```
const_reverse_iterator rbegin() const;
```

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

set::rend:

```
const_reverse_iterator rend() const;
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

set::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

set::swap:

```
void swap(set& x);
```

The member function swaps the controlled sequences between `*this` and `x`. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type `Pred`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

set::upper_bound:

```
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(key, x)` is true.

If no such element exists, the function returns `end()`.

set::value_comp:

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

Template functions

operator!=

```
template<class Key, class Pred, class A>
    bool operator!=(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
```

```
    bool operator!=(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class Key, class Pred, class A>
    bool operator==(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator==(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The first template function overloads `operator==` to compare two objects of template class multiset. The second template function overloads `operator==` to compare two objects of template class multiset. Both functions return `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end (), rhs.begin())`.

operator<

```
template<class Key, class Pred, class A>
    bool operator<(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The first template function overloads `operator<` to compare two objects of template class multiset. The second template function overloads `operator<` to compare two objects of template class multiset. Both functions return `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class Key, class Pred, class A>
    bool operator<=(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<=(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class Key, class Pred, class A>
    bool operator>(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class Key, class Pred, class A>
    bool operator>=(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>=(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class Key, class Pred, class A>
    void swap(
        multiset <Key, Pred, A>& lhs,
        multiset <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    void swap(
        set <Key, Pred, A>& lhs,
        set <Key, Pred, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

<sstream>

Description

Include the iostreams standard header **<sstream>** to define several template classes that support iostreams operations on sequences stored in an allocated array object. Such sequences are easily converted to and from objects of template class basic_string.

Synopsis

```
namespace std {
template<class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_stringbuf;
typedef basic_stringbuf<char> stringbuf;
typedef basic_stringbuf<wchar_t> wstringbuf;
template<class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_istringstream;
typedef basic_istringstream<char> istringstream;
typedef basic_istringstream<wchar_t> wistringstream;
template<class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_ostringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
template<class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_stringstream;
typedef basic_stringstream<char> stringstream;
typedef basic_stringstream<wchar_t> wstringstream;
    }
```

Classes

basic_stringbuf

Description: The template class describes a **stream buffer** that controls the transmission of elements of type E, whose character traits are determined by the class T, to and from a sequence of elements stored in an array object. The object is allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class basic_stringbuf<E, T, A> stores a copy of the ios_base::openmode argument from its constructor as its **stringbuf mode** mode:
- If mode & ios_base::in is nonzero, the input buffer is accessible.
- If mode & ios_base::out is nonzero, the output buffer is accessible.

Synopsis:

```
template <class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_stringbuf
        : public basic_streambuf<E, T> {
public:
    typedef typename basic_streambuf<E, T>::char_type
        char_type;
    typedef typename basic_streambuf<E, T>::traits_type
        traits_type;
    typedef typename basic_streambuf<E, T>::int_type
        int_type;
    typedef typename basic_streambuf<E, T>::pos_type
        pos_type;
    typedef typename basic_streambuf<E, T>::off_type
        off_type;
    basic_stringbuf(ios_base::openmode mode =
        ios_base::in | ios_base::out);
    basic_stringbuf(basic_string<E, T, A>& x,
        ios_base::openmode mode =
            ios_base::in | ios_base::out);
    basic_string<E, T, A> str() const;
    void str(basic_string<E, T, A>& x);
protected:
    virtual pos_type seekoff(off_type off,
        ios_base::seekdir way,
        ios_base::openmode mode =
            ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode mode =
            ios_base::in | ios_base::out);
    virtual int_type underflow();
    virtual int_type pbackfail(int_type c =
        traits_type::eof());
    virtual int_type overflow(int_type c =
        traits_type::eof());
    };
```

Constructor:

basic_stringbuf::basic_stringbuf:

```
basic_stringbuf(ios_base::openmode mode =
    ios_base::in | ios_base::out);
basic_stringbuf(basic_string<E, T, A>& x,
    ios_base::openmode mode =
        ios_base::in | ios_base::out);
```

The first constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. It also stores `mode` as the stringbuf mode.

The second constructor allocates a copy of the sequence controlled by the string object `x`. If `mode & ios_base::in` is nonzero, it sets the input buffer to begin reading at the start of the sequence. If `mode & ios_base::out` is nonzero, it sets the output buffer to begin writing at the start of the sequence. It also stores `mode` as the stringbuf mode.

Types:

basic_stringbuf::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_stringbuf::int_type:

```
typedef typename traits_type::int_type int_type;
```

The type is a synonym for `traits_type::int_type`.

basic_stringbuf::off_type:

```
typedef typename traits_type::off_type off_type;
```

The type is a synonym for `traits_type::off_type`.

basic_stringbuf::pos_type:

```
typedef typename traits_type::pos_type pos_type;
```

The type is a synonym for `traits_type::pos_type`.

basic_stringbuf::traits_type:

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

Member functions:

basic_stringbuf::overflow:

```
virtual int_type overflow(int_type c =
    traits_type::eof());
```

If `c` does not compare equal to `traits_type::eof()`, the protected virtual member function endeavors to insert the element `traits_type::to_char_type(c)` into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer. (Extending the output buffer this way also extends any associated input buffer.)

If the function cannot succeed, it returns `traits_type::eof()`. Otherwise, it returns `traits_type::not_eof(c)`.

basic_stringbuf::pbackfail:

```
virtual int_type pbackfail(int_type c =
    traits_type::eof());
```

The protected virtual member function endeavors to put back an element into the input buffer, then make it the current element (pointed to by the next pointer). If `c` compares equal to `traits_type::eof()`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `x = traits_type::to_char_type(c)`. The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to `x`, it can simply decrement the next pointer for the input buffer.
- If a putback position is available, and if the stringbuf mode permits the sequence to be altered (`mode & ios_base::out` is nonzero), it can store `x` into the putback position and decrement the next pointer for the input buffer.

If the function cannot succeed, it returns `traits_type::eof()`. Otherwise, it returns `traits_type::not_eof(c)`.

basic_stringbuf::seekoff:

```
virtual pos_type seekoff(off_type off,
    ios_base::seekdir way,
    ios_base::openmode mode =
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_stringbuf<E, T, A>`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If `way == ios_base::beg`, the new position is the beginning of the stream plus `off`.
- If `way == ios_base::cur`, the new position is the current stream position plus `off`.
- If `way == ios_base::end`, the new position is the end of the stream plus `off`.

If `mode & ios_base::in` is nonzero, the function alters the next position to read in the input buffer. If `mode & ios_base::out` is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. If the function affects both stream positions, `way` must be `ios_base::beg` or `ios_base::end` and both streams are positioned at the same element. Otherwise (or if neither position is affected) the positioning operation fails.

If the function succeeds in altering the stream position(s), it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

basic_stringbuf::seekpos:

```
virtual pos_type seekpos(pos_type sp,
    ios_base::openmode mode =
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_stringbuf<E, T, A>`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by `sp`.

If `mode & ios_base::in` is nonzero, the function alters the next position to read in the input buffer. If `mode & ios_base::out` is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. Otherwise (or if neither position is affected) the positioning operation fails.

If the function succeeds in altering the stream position(s), it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

basic_stringbuf::str:

```
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
```

The first member function returns an object of class `basic_string<E, T, A>`, whose controlled sequence is a copy of the sequence controlled by `*this`. The sequence copied depends on the stored stringbuf mode `mode`:

- If `mode & ios_base::out` is nonzero and an output buffer exists, the sequence is the entire output buffer (`epptr() - pbase()` elements beginning with `pbase()`).
- Otherwise, if `mode & ios_base::in` is nonzero and an input buffer exists, the sequence is the entire input buffer (`egptr() - eback()` elements beginning with `eback()`).
- Otherwise, the copied sequence is empty.

The second member function deallocates any sequence currently controlled by `*this`. It then allocates a copy of the sequence controlled by `x`. If `mode & ios_base::in` is nonzero, it sets the input buffer to begin reading at the beginning of the sequence. If `mode & ios_base::out` is nonzero, it sets the output buffer to begin writing at the beginning of the sequence.

basic_stringbuf::underflow:

```
virtual int_type underflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input buffer, then advance the current stream position, and return the element as `traits_type::to_int_type(c)`. It can do so in only one way: If a read position is available, it takes `c` as the element stored in the read position and advances the next pointer for the input buffer.

If the function cannot succeed, it returns `traits_type::eof()`. Otherwise, it returns the current element in the input stream, converted as described above.

basic_istringstream

Description: The template class describes an object that controls extraction of elements and encoded objects from a stream buffer of class `basic_stringbuf<E, T, A>`, with elements of type `E`, whose character traits are determined by the class `T`, and whose elements are allocated by an allocator of class `A`. The object stores an object of class `basic_stringbuf<E, T, A>`.

Synopsis:

```
template <class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_istringstream
        : public basic_istream<E, T> {
public:
    explicit basic_istringstream(
        ios_base::openmode mode = ios_base::in);
    explicit basic_istringstream(
        const basic_string<E, T, A>& x,
        ios_base::openmode mode = ios_base::in);
    basic_stringbuf<E, T, A> *rdbuf() const;
    basic_string<E, T, A>& str();
    void str(const basic_string<E, T, A>& x);
    };
```

Constructor:

basic_istringstream::basic_istringstream:

```
explicit basic_istringstream(
    ios_base::openmode mode = ios_base::in);
explicit basic_istringstream(
    const basic_string<E, T, A>& x,
    ios_base::openmode mode = ios_base::in);
```

The first constructor initializes the base class by calling `basic_istream(sb)`, where `sb` is the stored object of class `basic_stringbuf<E, T, A>`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(mode | ios_base::in)`.

The second constructor initializes the base class by calling `basic_istream(sb)`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(x, mode | ios_base::in)`.

Member functions:

basic_istringstream::rdbuf:

```
basic_stringbuf<E, T, A> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `basic_stringbuf<E, T, A>`.

basic_istringstream::str:

```
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
```

The first member function returns `rdbuf()-> str()`. The second member function calls `rdbuf()-> str(x)`.

basic_ostringstream

Description: The template class describes an object that controls insertion of elements and encoded objects into a stream buffer of class `basic_stringbuf<E, T, A>`, with elements of type `E`, whose character traits are determined by the class `T`, and whose elements are allocated by an allocator of class `A`. The object stores an object of class `basic_stringbuf<E, T, A>`.

Synopsis:

```
template <class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_ostringstream
```

```
        : public basic_ostream<E, T> {
public:
    explicit basic_ostringstream(
        ios_base::openmode mode = ios_base::out);
    explicit basic_ostringstream(
        const basic_string<E, T, A>& x,
        ios_base::openmode mode = ios_base::out);
    basic_stringbuf<E, T, A> *rdbuf() const;
    basic_string<E, T, A>& str();
    void str(const basic_string<E, T, A>& x);
    };
```

Constructor:

basic_ostringstream::basic_ostringstream:

```
explicit basic_ostringstream(
    ios_base::openmode mode = ios_base::out);
explicit basic_ostringstream(
    const basic_string<E, T, A>& x,
    ios_base::openmode mode = ios_base::out);
```

The first constructor initializes the base class by calling `basic_ostream(sb)`, where `sb` is the stored object of class `basic_stringbuf<E, T, A>`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(mode | ios_base::out)`.

The second constructor initializes the base class by calling `basic_ostream(sb)`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(x, mode | ios_base::out)`.

Member functions:

basic_ostringstream::rdbuf:

```
basic_stringbuf<E, T, A> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `basic_stringbuf<E, T, A>`.

basic_ostringstream::str:

```
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
```

The first member function returns `rdbuf()-> str()`. The second member function calls `rdbuf()-> str(x)`.

basic_stringstream

Description: The template class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class `basic_stringbuf<E, T, A>`, with elements of type `E`, whose character traits are determined by the class `T`, and whose elements are allocated by an allocator of class `A`. The object stores an object of class `basic_stringbuf<E, T, A>`.

Synopsis:

```
template <class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_stringstream
        : public basic_iostream<E, T> {
public:
    explicit basic_stringstream(
        ios_base::openmode mode =
```

```
            ios_base::in | ios_base::out);
    explicit basic_stringstream(
        const basic_string<E, T, A>& x,
        ios_base::openmode mode =
            ios_base::in | ios_base::out);
    basic_stringbuf<E, T, A> *rdbuf() const;
    basic_string<E, T, A>& str();
    void str(const basic_string<E, T, A>& x);
    };
```

Constructor:

basic_stringstream::basic_stringstream:

```
explicit basic_stringstream(
    ios_base::openmode mode =
        ios_base::in | ios_base::out);
explicit basic_stringstream(
    const basic_string<E, T, A>& x,
    ios_base::openmode mode =
         ios_base::in | ios_base::out);
```

The first constructor initializes the base class by calling `basic_iostream(sb)`, where `sb` is the stored object of class `basic_stringbuf<E, T, A>`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(mode)`.

The second constructor initializes the base class by calling `basic_ostream(sb)`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(x, mode)`.

Member functions:

basic_stringstream::rdbuf:

```
basic_stringbuf<E, T, A> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `basic_stringbuf<E, T, A>`.

basic_stringstream::str:

```
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
```

The first member function returns `rdbuf()-> str()`. The second member function calls `rdbuf()-> str(x)`.

Types

istringstream

```
typedef basic_istringstream<char> istringstream;
```

The type is a synonym for template class basic_istringstream, specialized for elements of type *char.*

ostringstream

```
typedef basic_ostringstream<char> ostringstream;
```

The type is a synonym for template class basic_ostringstream, specialized for elements of type *char.*

stringbuf

```
typedef basic_stringbuf<char> stringbuf;
```

The type is a synonym for template class basic_stringbuf, specialized for elements of type *char*.

stringstream

```
typedef basic_stringstream<char> stringstream;
```

The type is a synonym for template class `basic_stringstream`, specialized for elements of type *char*.

wistringstream

```
typedef basic_istringstream<wchar_t> wistringstream;
```

The type is a synonym for template class `basic_istringstream`, specialized for elements of type `wchar_t`.

wostringstream

```
typedef basic_ostringstream<wchar_t> wostringstream;
```

The type is a synonym for template class `basic_ostringstream`, specialized for elements of type `wchar_t`.

wstringbuf

```
typedef basic_stringbuf<wchar_t> wstringbuf;
```

The type is a synonym for template class `basic_stringbuf`, specialized for elements of type `wchar_t`.

wstringstream

```
typedef basic_stringstream<wchar_t> wstringstream;
```

The type is a synonym for template class `basic_stringstream`, specialized for elements of type `wchar_t`.

<stack>

Description

Include the STL standard header **<stack>** to define the template class `stack` and two supporting templates.

Synopsis

```
namespace std {
template<class T, class Cont>
    class stack;

        // TEMPLATE FUNCTIONS
template<class T, class Cont>
    bool operator==(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator!=(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator<(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
```

```
template<class T, class Cont>
    bool operator>(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator<=(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator>=(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
    }
```

Classes

stack

Description: The template class describes an object that controls a varying-length sequence of elements. The object allocates and frees storage for the sequence it controls through a protected object named **c**, of class `Cont`. The type `T` of elements in the controlled sequence must match value_type.

An object of class `Cont` must supply several public members defined the same as for deque, list, and vector (all of which are suitable candidates for class `Cont`). The required members are:

```
    typedef T value_type;
    typedef T0 size_type;
    Cont();
    bool empty() const;
    size_type size() const;
    value_type& back();
    const value_type& back() const;
    void push_back(const value_type& x);
    void pop_back();
 bool operator==(const Cont& X) const;
 bool operator!=(const Cont& X) const;
 bool operator<(const Cont& X) const;
 bool operator>(const Cont& X) const;
 bool operator<=(const Cont& X) const;
 bool operator>=(const Cont& X) const;
```

Here, `T0` is an unspecified type that meets the stated requirements.

Synopsis:

```
template<class T,
    class Cont = deque<T> >
    class stack {
public:
    typedef Cont container_type;
    typedef typename Cont::value_type value_type;
    typedef typename Cont::size_type size_type;
    stack();
    explicit stack(const container_type& cont);
    bool empty() const;
    size_type size() const;
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
protected:
    Cont c;
    };
```

Constructor:

stack::stack:

```
stack();
explicit stack(const container_type& cont);
```

The first constructor initializes the stored object with `c()`, to specify an empty initial controlled sequence. The second constructor initializes the stored object with `c(cont)`, to specify an initial controlled sequence that is a copy of the sequence controlled by `cont`.

Types:

stack::container_type:

```
typedef Cont container_type;
```

The type is a synonym for the template parameter `Cont`.

stack::size_type:

```
typedef typename Cont::size_type size_type;
```

The type is a synonym for `Cont::size_type`.

stack::value_type:

```
typedef typename Cont::value_type value_type;
```

The type is a synonym for `Cont::value_type`.

Member functions:

stack::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

stack::pop:

```
void pop();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

stack::push:

```
void push(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence.

stack::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

stack::top:

```
value_type& top();
const value_type& top() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

Template functions

operator!=

```
template<class T, class Cont>
    bool operator!=(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class Cont>
    bool operator==(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function overloads `operator==` to compare two objects of template class stack. The function returns `lhs.c == rhs.c`.

operator<

```
template<class T, class Cont>
    bool operator<(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function overloads `operator<` to compare two objects of template class stack. The function returns `lhs.c < rhs.c`.

operator<=

```
template<class T, class Cont>
    bool operator<=(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class Cont>
    bool operator>(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class Cont>
    bool operator>=(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function returns `!(lhs < rhs)`.

<stdexcept>

Description

Include the standard header **<stdexcept>** to define several classes used for reporting exceptions. The classes form a derivation hierarchy, as indicated by the indenting above, all derived from class exception.

Synopsis

```
namespace std {
class logic_error;
    class domain_error;
```

```
    class invalid_argument;
    class length_error;
    class out_of_range;

class runtime_error;
    class range_error;
    class overflow_error;
    class underflow_error;
    }
```

Classes

domain_error

```
class domain_error : public logic_error {
public:
    domain_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report a domain error. The value returned by `what()` is a copy of `what_arg.data()`.

invalid_argument

```
class invalid_argument : public logic_error {
public:
    invalid_argument(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report an invalid argument. The value returned by `what()` is a copy of `what_arg.data()`.

length_error

```
class length_error : public logic_error {
public:
    length_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report an attempt to generate an object too long to be specified. The value returned by `what()` is a copy of `what_arg.data()`.

logic_error

```
class logic_error : public exception {
public:
    logic_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report errors presumably detectable before the program executes, such as violations of logical preconditions. The value returned by `what()` is a copy of `what_arg.data()`.

out_of_range

```
class out_of_range : public logic_error {
public:
    out_of_range(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report an argument that is out of its valid range. The value returned by `what()` is a copy of `what_arg.data()`.

overflow_error

```
class overflow_error : public runtime_error {
public:
    overflow_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report an arithmetic overflow. The value returned by `what()` is a copy of `what_arg.data()`.

range_error

```
class range_error : public runtime_error {
public:
    range_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report a `range error`. The value returned by `what()` is a copy of `what_arg.data()`.

runtime_error

```
class runtime_error : public exception {
public:
    runtime_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report errors presumably detectable only when the program executes. The value returned by `what()` is a copy of `what_arg.data()`.

underflow_error

```
class underflow_error : public runtime_error {
public:
    underflow_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report an arithmetic underflow. The value returned by `what()` is a copy of `what_arg.data()`.

<streambuf>

Description

Include the iostreams standard header **<streambuf>** to define template class basic_streambuf, which is basic to the operation of the iostreams classes. (This header is typically included for you by another of the iostreams headers. You seldom have occasion to include it directly.)

Synopsis

```
namespace std {
template<class E, class T = char_traits<E> >
    class basic_streambuf;
typedef basic_streambuf<char, char_traits<char> >
    streambuf;
typedef basic_streambuf<wchar_t,
    char_traits<wchar_t> > wstreambuf;
    }
```

Classes

basic_streambuf

Description: The template class describes an abstract base class for deriving a **stream buffer**, which controls the transmission of elements to and from a specific representation of a stream. An object of class basic_streambuf helps control a stream with elements of type T, also known as char_type, whose character traits are determined by the class char_traits, also known as traits_type.

Every stream buffer conceptually controls two independent streams, in fact, one for extractions (input) and one for insertions (output). A specific representation may, however, make either or both of these streams inaccessible. It typically maintains some relationship between the two streams. What you insert into the output stream of a basic_stringbuf<E, T> object, for example, is what you later extract from its input stream. And when you position one stream of a basic_filebuf<E, T> object, you position the other stream in tandem.

The public interface to template class basic_streambuf supplies the operations common to all stream buffers, however specialized. The protected interface supplies the operations needed for a specific representation of a stream to do its work. The protected virtual member functions let you tailor the behavior of a derived stream buffer for a specific representation of a stream. Each of the derived stream buffers in this library describes how it specializes the behavior of its protected virtual member functions. Documented here is the **default behavior** for the base class, which is often to do nothing.

The remaining protected member functions control copying to and from any storage supplied to buffer transmissions to and from streams. An **input buffer**, for example, is characterized by:
- eback(), a pointer to the beginning of the buffer
- gptr(), a pointer to the next element to read
- egptr(), a pointer just past the end of the buffer

Similarly, an **output buffer** is characterized by:
- pbase(), a pointer to the beginning of the buffer
- pptr(), a pointer to the next element to write
- epptr(), a pointer just past the end of the buffer

For any buffer, the protocol is:
- If the next pointer is null, no buffer exists. Otherwise, all three pointers point into the same sequence. (They can be safely compared for order.)
- For an output buffer, if the next pointer compares less than the end pointer, you can store an element at the **write position** designated by the next pointer.
- For an input buffer, if the next pointer compares less than the end pointer, you can read an element at the **read position** designated by the next pointer.
- For an input buffer, if the beginning pointer compares less than the next pointer, you can put back an element at the **putback position** designated by the decremented next pointer.

Any protected virtual member functions you write for a class derived from basic_streambuf<E, T> must cooperate in maintaining this protocol.

An object of class basic_streambuf<E, T> stores the six pointers described above. It also stores a **locale object** in an object of type locale for potential use by a derived stream buffer.

Synopsis:

```
template <class E, class T = char_traits<E> >
    class basic_streambuf {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef typename traits_type::int_type int_type;
    typedef typename traits_type::pos_type pos_type;
    typedef typename traits_type::off_type off_type;
    virtual ~streambuf();
    locale pubimbue(const locale& loc);
    locale getloc() const;
    basic_streambuf *pubsetbuf(char_type *s,
        streamsize n);
    pos_type pubseekoff(off_type off,
        ios_base::seekdir way,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    int pubsync();
    streamsize in_avail();
    int_type snextc();
    int_type sbumpc();
    int_type sgetc();
    void stossc();  // OPTIONAL
    streamsize sgetn(char_type *s, streamsize n);
    int_type sputbackc(char_type c);
    int_type sungetc();
    int_type sputc(char_type c);
    streamsize sputn(const char_type *s, streamsize n);
protected:
    basic_streambuf();
    char_type *eback() const;
    char_type *gptr() const;
    char_type *egptr() const;
    void gbump(int n);
    void setg(char_type *gbeg,
        char_type *gnext, char_type *gend);
    char_type *pbase() const;
    char_type *pptr() const;
    char_type *epptr() const;
    void pbump(int n);
    void setp(char_type *pbeg, char_type *pend);
    virtual void imbue(const locale &loc);
    virtual basic_streambuf *setbuf(char_type *s,
        streamsize n);
    virtual pos_type seekoff(off_type off,
        ios_base::seekdir way,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    virtual int sync();
    virtual streamsize showmanyc();
    virtual streamsize xsgetn(char_type *s,
        streamsize n);
    virtual int_type underflow();
    virtual int_type uflow();
    virtual int_type pbackfail(int_type c =
```

```
        traits_type::eof());
    virtual streamsize xsputn(const char_type *s,
        streamsize n);
    virtual int_type overflow(int_type c =
        traits_type::eof());
    };
```

Constructor:

basic_streambuf::basic_streambuf:

```
basic_streambuf();
```

The protected constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. It also stores `locale::classic()` in the locale object.

Types:

basic_streambuf::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_streambuf::int_type:

```
typedef typename traits_type::int_type int_type;
```

The type is a synonym for `traits_type::int_type`.

basic_streambuf::off_type:

```
typedef typename traits_type::off_type off_type;
```

The type is a synonym for `traits_type::off_type`.

basic_streambuf::pos_type:

```
typedef typename traits_type::pos_type pos_type;
```

The type is a synonym for `traits_type::pos_type`.

basic_streambuf::traits_type:

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

Member functions:

basic_streambuf::eback:

```
char_type *eback() const;
```

The member function returns a pointer to the beginning of the input buffer.

basic_streambuf::egptr:

```
char_type *egptr() const;
```

The member function returns a pointer just past the end of the input buffer.

basic_streambuf::epptr:

```
char_type *epptr() const;
```

The member function returns a pointer just past the end of the output buffer.

basic_streambuf::gbump:

```
void gbump(int n);
```

The member function adds n to the next pointer for the input buffer.

basic_streambuf::getloc:

```
locale getloc() const;
```

The member function returns the stored locale object.

basic_streambuf::gptr:

```
char_type *gptr() const;
```

The member function returns a pointer to the next element of the input buffer.

basic_streambuf::imbue:

```
virtual void imbue(const locale &loc);
```

The default behavior is to do nothing.

basic_streambuf::in_avail:

```
streamsize in_avail();
```

If a read position is available, the member function returns `egptr() - gptr()`. Otherwise, it returns `showmanyc()`.

basic_streambuf::overflow:

```
virtual int_type overflow(int_type c =
    traits_type::eof());
```

If `c` does not compare equal to `traits_type::eof()`, the protected virtual member function endeavors to insert the element `traits_type:: to_char_type(c)` into the output stream. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer.
- It can make a write position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

If the function cannot succeed, it returns `traits_type::eof()` or throws an exception. Otherwise, it returns `traits_type::not_eof(c)`. The default behavior is to return `traits_type::eof()`.

basic_streambuf::pbackfail:

```
virtual int_type pbackfail(int_type c =
    traits_type::eof());
```

The protected virtual member function endeavors to put back an element into the input stream, then make it the current element (pointed to by the next pointer). If `c` compares equal to `traits_type::eof()`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `traits_type::to_char_type(c)`. The function can put back an element in various ways:

- If a putback position is available, it can store the element into the putback position and decrement the next pointer for the input buffer.
- It can make a putback position available by allocating new or additional storage for the input buffer.
- For a stream buffer with common input and output streams, it can make a putback position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

If the function cannot succeed, it returns `traits_type::eof()` or throws an exception. Otherwise, it returns some other value. The default behavior is to return `traits_type::eof()`.

basic_streambuf::pbase:

```
char_type *pbase() const;
```

The member function returns a pointer to the beginning of the output buffer.

basic_streambuf::pbump:

```
void pbump(int n);
```

The member function adds `n` to the next pointer for the output buffer.

basic_streambuf::pptr:

```
char_type *pptr() const;
```

The member function returns a pointer to the next element of the output buffer.

basic_streambuf::pubimbue:

```
locale pubimbue(const locale& loc);
```

The member function stores `loc` in the locale object, calls `imbue()`, then returns the previous value stored in the locale object.

basic_streambuf::pubseekoff:

```
pos_type pubseekoff(off_type off,
    ios_base::seekdir way,
    ios_base::openmode which =
        ios_base::in | ios_base::out);
```

The member function returns `seekoff(off, way, which)`.

basic_streambuf::pubseekpos:

```
pos_type pubseekpos(pos_type sp,
    ios_base::openmode which =
        ios_base::in | ios_base::out);
```

The member function returns `seekpos(sp, which)`.

basic_streambuf::pubsetbuf:

```
basic_streambuf *pubsetbuf(char_type *s, streamsize n);
```

The member function returns stbuf(s, n).

basic_streambuf::pubsync:

```
int pubsync();
```

The member function returns sync().

basic_streambuf::sbumpc:

```
int_type sbumpc();
```

If a read position is available, the member function returns traits_type::to_int_type(*gptr()) and increments the next pointer for the input buffer. Otherwise, it returns uflow().

basic_streambuf::seekoff:

```
virtual pos_type seekoff(off_type off,
    ios_base::seekdir way,
    ios_base::openmode which =
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. The new position is determined as follows:

- If way == ios_base::beg, the new position is the beginning of the stream plus off.
- If way == ios_base::cur, the new position is the current stream position plus off.
- If way == ios_base::end, the new position is the end of the stream plus off.

Typically, if which & ios_base::in is nonzero, the input stream is affected, and if which & ios_base::out is nonzero, the output stream is affected. Actual use of this parameter varies among derived stream buffers, however.

If the function succeeds in altering the stream position(s), it returns the resultant stream position (or one of them). Otherwise, it returns an invalid stream position. The default behavior is to return an invalid stream position.

basic_streambuf::seekpos:

```
virtual pos_type seekpos(pos_type sp,
    ios_base::openmode which =
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. The new position is sp.

Typically, if which & ios_base::in is nonzero, the input stream is affected, and if which & ios_base::out is nonzero, the output stream is affected. Actual use of this parameter varies among derived stream buffers, however.

If the function succeeds in altering the stream position(s), it returns the resultant stream position (or one of them). Otherwise, it returns an invalid stream position. The default behavior is to return an invalid stream position.

basic_streambuf::setbuf:

```
virtual basic_streambuf *setbuf(char_type *s,
    streamsize n);
```

The protected virtual member function performs an operation peculiar to each derived stream buffer. (See, for example, basic_filebuf.) The default behavior is to return `this`.

basic_streambuf::setg:

```
void setg(char_type *gbeg, char_type *gnext,
    char_type *gend);
```

The member function stores `gbeg` in the beginning pointer, `gnext` in the next pointer, and `gend` in the end pointer for the input buffer.

basic_streambuf::setp:

```
void setp(char_type *pbeg, char_type *pend);
```

The member function stores `pbeg` in the beginning pointer, `pbeg` in the next pointer, and `pend` in the end pointer for the output buffer.

basic_streambuf::sgetc:

```
int_type sgetc();
```

If a read position is available, the member function returns `traits_type::to_int_type( *gptr())` Otherwise, it returns `underflow()`.

basic_streambuf::sgetn:

```
streamsize sgetn(char_type *s, streamsize n);
```

The member function returns `xsgetn(s, n)`.

basic_streambuf::showmanyc:

```
virtual streamsize showmanyc();
```

The protected virtual member function returns a count of the number of characters that can be extracted from the input stream with no fear that the program will suffer an indefinite wait. The default behavior is to return zero.

basic_streambuf::snextc:

```
int_type snextc();
```

The member function calls `sbumpc()` and, if that function returns `traits_type::eof()`, returns `traits_type::eof()`. Otherwise, it returns `sgetc()`.

basic_streambuf::sputbackc:

```
int_type sputbackc(char_type c);
```

If a putback position is available and `c` compares equal to the character stored in that position, the member function decrements the next pointer for the input buffer and returns `ch`, which is the value `traits_type::to_int_type(c)`. Otherwise, it returns `pbackfail(ch)`.

basic_streambuf::sputc:

```
int_type sputc(char_type c);
```

If a write position is available, the member function stores `c` in the write position, increments the next pointer for the output buffer, and returns `ch`, which is the value `traits_type::to_int_type(c)`. Otherwise, it returns `overflow(ch)`.

basic_streambuf::sputn:

```
streamsize sputn(const char_type *s, streamsize n);
```

The member function returns `xsputn(s, n)`.

basic_streambuf::stossc:

```
void stossc();  // OPTIONAL
```

The member function calls `sbumpc()`. Note that an implementation is not required to supply this member function.

basic_streambuf::sungetc:

```
int_type sungetc();
```

If a putback position is available, the member function decrements the next pointer for the input buffer and returns `traits_type::to_int_type( *gptr())`. Otherwise it returns `pbackfail()`.

basic_streambuf::sync:

```
virtual int sync();
```

The protected virtual member function endeavors to synchronize the controlled streams with any associated external streams. Typically, this involves writing out any elements between the beginning and next pointers for the output buffer. It does *not* involve putting back any elements between the next and end pointers for the input buffer. If the function cannot succeed, it returns -1. The default behavior is to return zero.

basic_streambuf::uflow:

```
virtual int_type uflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input stream, then advance the current stream position, and return the element as `traits_type::to_int_type(c)`. It can do so in various ways:

- If a read position is available, it takes `c` as the element stored in the read position and advances the next pointer for the input buffer.
- It can read an element directly, from some external source, and deliver it as the value `c`.
- For a stream buffer with common input and output streams, it can make a read position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer. Or it can allocate new or additional storage for the input buffer. The function then reads in, from some external source, one or more elements.

If the function cannot succeed, it returns `traits_type::eof()`, or throws an exception. Otherwise, it returns the current element `c` in the input stream, converted as described above, and advances the next pointer for the input buffer. The default behavior is to call `underflow()` and, if that function returns `traits_type::eof()`, to return `traits_type::eof()`. Otherwise, the function returns the current element `c` in the input stream, converted as described above, and advances the next pointer for the input buffer.

basic_streambuf::underflow:

```
virtual int_type underflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input stream, without advancing the current stream position, and return it as `traits_type::to_int_type(c)`. It can do so in various ways:

- If a read position is available, `c` is the element stored in the read position.
- It can make a read position available by allocating new or additional storage for the input buffer, then reading in, from some external source, one or more elements.

If the function cannot succeed, it returns `traits_type::eof()`, or throws an exception. Otherwise, it returns the current element in the input stream, converted as described above. The default behavior is to return `traits_type::eof()`.

basic_streambuf::xsgetn:

```
virtual streamsize xsgetn(char_type *s, streamsize n);
```

The protected virtual member function extracts up to `n` elements from the input stream, as if by repeated calls to sbumpc, and stores them in the array beginning at `s`. It returns the number of elements actually extracted.

basic_streambuf::xsputn:

```
virtual streamsize xsputn(const char_type *s,
    streamsize n);
```

The protected virtual member function inserts up to `n` elements into the output stream, as if by repeated calls to sputc, from the array beginning at `s`. It returns the number of elements actually inserted.

Types

streambuf

```
typedef basic_streambuf<char, char_traits<char> >
    streambuf;
```

The type is a synonym for template class `basic_streambuf`, specialized for elements of type *char* with default character traits.

wstreambuf

```
typedef basic_streambuf<wchar_t, char_traits<wchar_t> >
    wstreambuf;
```

The type is a synonym for template class `basic_streambuf`, specialized for elements of type `wchar_t` with default character traits.

<string>

Description

Include the standard header **<string>** to define the container template class basic_string and various supporting templates.

Synopsis

```
namespace std {
template<class E>
    class char_traits;
template<>
    class char_traits<char>;
```

```
template<>
    class char_traits<wchar_t>;
template<class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_string;
typedef basic_string<char> string;
typedef basic_string>wchar_t> wstring;

        // TEMPLATE FUNCTIONS
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        E rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        E lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator==(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator!=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator<(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
```

```
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator<=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    void swap(
        basic_string<E, T, A>& lhs,
        basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_ostream<E>& operator<<(
        basic_ostream<E>& os,
        const basic_string<E, T, A>& str);
template<class E, class T, class A>
    basic_istream<E>& operator>>(
        basic_istream<E>& is,
        basic_string<E, T, A>& str);
template<class E, class T, class A>
    basic_istream<E, T>& getline(
        basic_istream<E, T>& is,
        basic_string<E, T, A>& str);
 template<class E, class T, class A>
    basic_istream<E, T>& getline(
        basic_istream<E, T>& is,
        basic_string<E, T, A>& str,
        E delim);
    }
```

Classes

basic_string

Description: The template class describes an object that controls a varying-length sequence of elements of type E, also known as value_type. Such an element type must not require explicit construction or destruction, and it must be suitable for use as the E parameter to basic_istream or basic_ostream (A "plain old data structure", or **POD**, from C generally meets this criterion.) The Standard C++ Library provides two specializations of this template class, with the type definitions string, for elements of type *char,* and wstring, for elements of type wchar_t.

Various important properties of the elements in a basic_string specialization are described by the class T, also known as traits_type. A class that specifies these character traits must have the same external interface as an object of template class char_traits.

The object allocates and frees storage for the sequence it controls through a stored allocator object of class A, also known as allocator_type. Such an allocator object must have the same external interface as an object of template class allocator. (Class char_traits has no provision for alternate addressing schemes, such as might be required to implement a far heap.) Note that the stored allocator object is *not* copied when the container object is assigned.

The sequences controlled by an object of template class basic_string are usually called **strings**. These objects should not be confused, however, with the null-terminated C strings used throughout the Standard C++ Library.

Many member functions require an **operand sequence** of elements. You can specify such an operand sequence several ways:

- c — one element with value c
- n, c — a repetition of n elements each with value c
- s — a null-terminated sequence (such as a C string, for E of type *char*) beginning at s (which must not be a null pointer), where the terminating element is the value value_type() and is not part of the operand sequence
- s, n — a sequence of n elements beginning at s (which must not be a null pointer)
- str — the sequence specified by the basic_string object str
- str, pos, n — the substring of the basic_string object str with up to n elements (or through the end of the string, whichever comes first) beginning at position pos
- first, last — a sequence of elements delimited by the iterators first and last, in the range [first, last), which must *not* overlap the sequence controlled by the string object whose member function is being called

If a **position argument** (such as pos above) is beyond the end of the string on a call to a basic_string member function, the function reports an **out-of-range error** by throwing an object of class out_of_range.

If a function is asked to generate a sequence longer than max_size() elements, the function reports a **length error** by throwing an object of class length_error.

References, pointers, and iterators that designate elements of the controlled sequence can become invalid after any call to a function that alters the controlled sequence, or after the first call to the non-const member functions at, begin, end , operator[], rbegin, or rend. (The idea is to permit multiple strings to share the same representation until one string becomes a candidate for change, at which point that string makes a private copy of the representation, using a discipline called **copy on write**.)

Synopsis:

```
template<class E,
    class T = char_traits<E>,
    class A = allocator<T> >
    class basic_string {
public:
    typedef T traits_type;
    typedef A allocator_type;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator>
        reverse_iterator;
    typedef typename allocator_type::pointer
        pointer;
    typedef typename allocator_type::const_pointer
        const_pointer;
    typedef typename allocator_type::reference
        reference;
    typedef typename allocator_type::const_reference
        const_reference;
    typedef typename allocator_type::value_type
        value_type;
    static const size_type npos = -1;
    basic_string();
    explicit basic_string(const allocator_type& al);
    basic_string(const basic_string& rhs);
    basic_string(const basic_string& rhs, size_type pos,
        size_type n = npos);
    basic_string(const basic_string& rhs, size_type pos,
        size_type n, const allocator_type& al);
    basic_string(const value_type *s, size_type n);
    basic_string(const value_type *s, size_type n,
        const allocator_type& al);
    basic_string(const value_type *s);
    basic_string(const value_type *s,
        const allocator_type& al);
    basic_string(size_type n, value_type c);
    basic_string(size_type n, value_type c,
        const allocator_type& al);
    template <class InIt>
        basic_string(InIt first, InIt last);
    template <class InIt>
        basic_string(InIt first, InIt last,
            const allocator_type& al);
    basic_string& operator=(const basic_string& rhs);
    basic_string& operator=(const value_type *s);
    basic_string& operator=(value_type c);
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    const_reference at(size_type pos) const;
    reference at(size_type pos);
    const_reference operator[](size_type pos) const;
    reference operator[](size_type pos);
    void push_back(value_type c);
    const value_type *c_str() const;
    const value_type *data() const;
    size_type length() const;
```

```
size_type size() const;
size_type max_size() const;
void resize(size_type n, value_type c = value_type());
size_type capacity() const;
void reserve(size_type n = 0);
bool empty() const;
basic_string& operator+=(const basic_string& rhs);
basic_string& operator+=(const value_type *s);
basic_string& operator+=(value_type c);

basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str,
    size_type pos, size_type n);
basic_string& append(const value_type *s,
    size_type n);
basic_string& append(const value_type *s);
basic_string& append(size_type n, value_type c);
template<class InIt>
    basic_string& append(InIt first, InIt last);
basic_string& assign(const basic_string& str);
basic_string& assign(const basic_string& str,
    size_type pos, size_type n);
basic_string& assign(const value_type *s,
    size_type n);
basic_string& assign(const value_type *s);
basic_string& assign(size_type n, value_type c);
template<class InIt>
    basic_string& assign(InIt first, InIt last);
basic_string& insert(size_type p0,
    const basic_string& str);
basic_string& insert(size_type p0,
    const basic_string& str, size_type pos,
        size_type n);
basic_string& insert(size_type p0,
    const value_type *s, size_type n);
basic_string& insert(size_type p0,
    const value_type *s);
basic_string& insert(size_type p0,
    size_type n, value_type c);
iterator insert(iterator it,
    value_type c = value_type());
void insert(iterator it, size_type n, value_type c);
template<class InIt>
    void insert(iterator it,
        InIt first, InIt last);
basic_string& erase(size_type p0 = 0,
    size_type n = npos);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
void clear();
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str);
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str, size_type pos,
        size_type n);
basic_string& replace(size_type p0, size_type n0,
    const value_type *s, size_type n);
basic_string& replace(size_type p0, size_type n0,
    const value_type *s);
basic_string& replace(size_type p0, size_type n0,
    size_type n, value_type c);
basic_string& replace(iterator first0, iterator last0,
    const basic_string& str);
basic_string& replace(iterator first0, iterator last0,
    const value_type *s, size_type n);
basic_string& replace(iterator first0, iterator last0,
    const value_type *s);
basic_string& replace(iterator first0, iterator last0,
```

```
    size_type n, value_type c);
template<class InIt>
    basic_string&
        replace(iterator first0, iterator last0,
            InIt first, InIt last);
size_type copy(value_type *s, size_type n,
    size_type pos = 0) const;
void swap(basic_string& str);
size_type find(const basic_string& str,
    size_type pos = 0) const;
size_type find(const value_type *s, size_type pos,
    size_type n) const;
size_type find(const value_type *s,
    size_type pos = 0) const;
size_type find(value_type c, size_type pos = 0) const;
size_type rfind(const basic_string& str,
    size_type pos = npos) const;
size_type rfind(const value_type *s, size_type pos,
    size_type n = npos) const;
size_type rfind(const value_type *s,
    size_type pos = npos) const;
size_type rfind(value_type c,
    size_type pos = npos) const;
size_type find_first_of(const basic_string& str,
    size_type pos = 0) const;
size_type find_first_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_first_of(const value_type *s,
    size_type pos = 0) const;
size_type find_first_of(value_type c,
    size_type pos = 0) const;
size_type find_last_of(const basic_string& str,
    size_type pos = npos) const;
size_type find_last_of(const value_type *s,
    size_type pos, size_type n = npos) const;
size_type find_last_of(const value_type *s,
    size_type pos = npos) const;
size_type find_last_of(value_type c,
    size_type pos = npos) const;
size_type find_first_not_of(const basic_string& str,
    size_type pos = 0) const;
size_type find_first_not_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_first_not_of(const value_type *s,
    size_type pos = 0) const;
size_type find_first_not_of(value_type c,
    size_type pos = 0) const;
size_type find_last_not_of(const basic_string& str,
    size_type pos = npos) const;
size_type find_last_not_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_last_not_of(const value_type *s,
    size_type pos = npos) const;
size_type find_last_not_of(value_type c,
    size_type pos = npos) const;
basic_string substr(size_type pos = 0,
    size_type n = npos) const;
int compare(const basic_string& str) const;
int compare(size_type p0, size_type n0,
    const basic_string& str);
int compare(size_type p0, size_type n0,
    const basic_string& str, size_type pos,
        size_type n);
int compare(const value_type *s) const;
int compare(size_type p0, size_type n0,
    const value_type *s) const;
```

```
    int compare(size_type p0, size_type n0,
        const value_type *s, size_type pos) const;
    allocator_type get_allocator() const;
    };
```

Constructor:

basic_string::basic_string:

```
basic_string(const value_type *s);
basic_string(const value_type *s,
    const allocator_type& al);
basic_string(const value_type *s, size_type n);
basic_string(const value_type *s, size_type n,
    const allocator_type& al);
basic_string(const basic_string& rhs);
basic_string(const basic_string& rhs, size_type pos,
    size_type n = npos);
basic_string(const basic_string& rhs, size_type pos,
    size_type n, const allocator_type& al);
basic_string(size_type n, value_type c);
basic_string(size_type n, value_type c,
    const allocator_type& al);
basic_string();
explicit basic_string(const allocator_type& al);
template <class InIt>
    basic_string(InIt first, InIt last);
template <class InIt>
    basic_string(InIt first, InIt last, const allocator_type& al);
```

All constructors store an allocator object and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

The controlled sequence is initialized to a copy of the operand sequence specified by the remaining operands. A constructor with no operand sequence specifies an empty initial controlled sequence. If `InIt` is an integer type in a template constructor, the operand sequence `first, last` behaves the same as `(size_type)first, (value_type)last`.

In this implementation, if a translator does not support member template functions, the templates:

```
template <class InIt>
    basic_string(InIt first, InIt last);
template <class InIt>
    basic_string(InIt first, InIt last,
        const allocator_type& al);
```

are replaced by:

```
basic_string(const_pointer first, const_pointer last);
basic_string(const_pointer first, const_pointer last,
    const allocator_type& al);
```

Constants:

basic_string::npos:

```
static const size_type npos = -1;
```

The constant is the largest representable value of type `size_type`. It is assuredly larger than `max_size()`, hence it serves as either a very large value or as a special code.

Types:

basic_string::allocator_type:

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

basic_string::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

basic_string::const_pointer:

```
typedef typename allocator_type::const_pointer
    const_pointer;
```

The type is a synonym for `allocator_type::const_pointer`.

basic_string::const_reference:

```
typedef typename allocator_type::const_reference
    const_reference;
```

The type is a synonym for `allocator_type::const_reference`.

basic_string::const_reverse_iterator:

```
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse iterator for the controlled sequence.

basic_string::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

basic_string::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

basic_string::pointer:

```
typedef typename allocator_type::pointer
    pointer;
```

The type is a synonym for `allocator_type::pointer`.

basic_string::reference:

```
typedef typename allocator_type::reference
    reference;
```

The type is a synonym for `allocator_type::reference`.

basic_string::reverse_iterator:

```
typedef reverse_iterator<iterator>
    reverse_iterator;
```

The type describes an object that can serve as a reverse iterator for the controlled sequence.

basic_string::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type `T2`.

basic_string::traits_type:

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

basic_string::value_type:

```
typedef typename allocator_type::value_type
    value_type;
```

The type is a synonym for `allocator_type::value_type`.

Member functions:

basic_string::append:

```
basic_string& append(const value_type *s);
basic_string& append(const value_type *s,
    size_type n);
basic_string& append(const basic_string& str,
    size_type pos, size_type n);
basic_string& append(const basic_string& str);
basic_string& append(size_type n, value_type c);
template<class InIt>
    basic_string& append(InIt first, InIt last);
```

If `InIt` is an integer type, the template member function behaves the same as `append((size_type)first, (value_type)last)`. Otherwise, the member functions each append the operand sequence to the end of the sequence controlled by `*this`, then return `*this`.

In this implementation, if a translator does not support member template functions, the template:

```
template<class InIt>
    basic_string& append(InIt first, InIt last);
```

is replaced by:

```
basic_string& append(const_pointer first,
    const_pointer last);
```

basic_string::assign:

```
basic_string& assign(const value_type *s);
basic_string& assign(const value_type *s,
    size_type n);
```

```
basic_string& assign(const basic_string& str,
    size_type pos, size_type n);
basic_string& assign(const basic_string& str);
basic_string& assign(size_type n, value_type c);
template<class InIt>
    basic_string& assign(InIt first, InIt last);
```

If `InIt` is an integer type, the template member function behaves the same as `assign((size_type)first, (value_type)last)`. Otherwise, the member functions each replace the sequence controlled by `*this` with the operand sequence, then return `*this`.

In this implementation, if a translator does not support member template functions, the template:

```
template<class InIt>
    basic_string& assign(InIt first, InIt last);
```

is replaced by:

```
basic_string& assign(const_pointer first,
    const_pointer last);
```

basic_string::at:

```
const_reference at(size_type pos) const;
reference at(size_type pos);
```

The member functions each return a reference to the element of the controlled sequence at position `pos`, or report an out-of-range error.

basic_string::begin:

```
const_iterator begin() const;
iterator begin();
```

The member functions each return a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

basic_string::c_str:

```
const value_type *c_str() const;
```

The member function returns a pointer to a non-modifiable C string constructed by adding a terminating null element (`value_type()`) to the controlled sequence. Calling any non-const member function for `*this` can invalidate the pointer.

basic_string::capacity:

```
size_type capacity() const;
```

The member function returns the storage currently allocated to hold the controlled sequence, a value at least as large as `size()`.

basic_string::clear:

```
void clear();
```

The member function calls `erase( begin(), end())`.

basic_string::compare:

```
int compare(const basic_string& str) const;
int compare(size_type p0, size_type n0,
    const basic_string& str);
```

```
int compare(size_type p0, size_type n0,
    const basic_string& str, size_type pos, size_type n);
int compare(const value_type *s) const;
int compare(size_type p0, size_type n0,
    const value_type *s) const;
int compare(size_type p0, size_type n0,
    const value_type *s, size_type pos) const;
```

The member functions each compare up to `n0` elements of the controlled sequence beginning with position `p0`, or the entire controlled sequence if these arguments are not supplied, to the operand sequence. Each function returns:

- a negative value if the first differing element in the controlled sequence compares less than the corresponding element in the operand sequence (as determined by `traits_type::compare`), or if the two have a common prefix but the operand sequence is longer
- zero if the two compare equal element by element and are the same length
- a positive value otherwise

basic_string::copy:

```
size_type copy(value_type *s, size_type n,
    size_type pos = 0) const;
```

The member function copies up to `n` elements from the controlled sequence, beginning at position `pos`, to the array of `value_type` beginning at `s`. It returns the number of elements actually copied.

basic_string::data:

```
const value_type *data() const;
```

The member function returns a pointer to the first element of the sequence (or, for an empty sequence, a non-null pointer that cannot be dereferenced).

basic_string::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

basic_string::end:

```
const_iterator end() const;
iterator end();
```

The member functions each return a random-access iterator that points just beyond the end of the sequence.

basic_string::erase:

```
iterator erase(iterator first, iterator last);
iterator erase(iterator it);
basic_string& erase(size_type p0 = 0,
    size_type n = npos);
```

The first member function removes the elements of the controlled sequence in the range `[first, last)`. The second member function removes the element of the controlled sequence pointed to by `it`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member function removes up to `n` elements of the controlled sequence beginning at position `p0`, then returns `*this`.

basic_string::find:

```
size_type find(value_type c, size_type pos = 0) const;
size_type find(const value_type *s,
    size_type pos = 0) const;
size_type find(const value_type *s, size_type pos,
    size_type n) const;
size_type find(const basic_string& str,
    size_type pos = 0) const;
```

The member functions each find the first (lowest beginning position) subsequence in the controlled sequence, beginning on or after position `pos`, that matches the operand sequence specified by the remaining operands. If it succeeds, it returns the position where the matching subsequence begins. Otherwise, the function returns npos.

basic_string::find_first_not_of:

```
size_type find_first_not_of(value_type c,
    size_type pos = 0) const;
size_type find_first_not_of(const value_type *s,
    size_type pos = 0) const;
size_type find_first_not_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_first_not_of(const basic_string& str,
    size_type pos = 0) const;
```

The member functions each find the first (lowest position) element of the controlled sequence, at or after position `pos`, that matches *none* of the elements in the operand sequence specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns npos.

basic_string::find_first_of:

```
size_type find_first_of(value_type c,
    size_type pos = 0) const;
size_type find_first_of(const value_type *s,
    size_type pos = 0) const;
size_type find_first_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_first_of(const basic_string& str,
    size_type pos = 0) const;
```

The member functions each find the first (lowest position) element of the controlled sequence, at or after position `pos`, that matches *any* of the elements in the operand sequence specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns npos.

basic_string::find_last_not_of:

```
size_type find_last_not_of(value_type c,
    size_type pos = npos) const;
size_type find_last_not_of(const value_type *s,
    size_type pos = npos) const;
size_type find_last_not_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_last_not_of(const basic_string& str,
    size_type pos = npos) const;
```

The member functions each find the last (highest position) element of the controlled sequence, at or before position `pos`, that matches *none* of the elements in the operand sequence specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns npos.

basic_string::find_last_of:

```
size_type find_last_of(value_type c,
    size_type pos = npos) const;
size_type find_last_of(const value_type *s,
    size_type pos = npos) const;
size_type find_last_of(const value_type *s,
    size_type pos, size_type n = npos) const;
size_type find_last_of(const basic_string& str,
    size_type pos = npos) const;
```

The member functions each find the last (highest position) element of the controlled sequence, at or before position `pos`, that matches *any* of the elements in the operand sequence specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns npos.

basic_string::get_allocator:

```
allocator_type get_allocator() const;
```

The member function returns the stored allocator object.

basic_string::insert:

```
basic_string& insert(size_type p0, const value_type *s);
basic_string& insert(size_type p0, const value_type *s,
    size_type n);
basic_string& insert(size_type p0,
    const basic_string& str);
basic_string& insert(size_type p0,
    const basic_string& str, size_type pos, size_type n);
basic_string& insert(size_type p0,
    size_type n, value_type c);
iterator insert(iterator it,
    value_type c = value_type());
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
void insert(iterator it, size_type n, value_type c);
```

The member functions each insert, before position `p0` or before the element pointed to by `it` in the controlled sequence, the operand sequence specified by the remaining operands. A function that returns a value returns `*this`. If `InIt` is an integer type in the template member function, the operand sequence `first, last` behaves the same as `(size_type)first, (value_type)last`.

In this implementation, if a translator does not support member template functions, the template:

```
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
```

is replaced by:

```
void insert(iterator it,
    const_pointer first, const_pointer last);
```

basic_string::length:

```
size_type length() const;
```

The member function returns the length of the controlled sequence (same as `size()`).

basic_string::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

basic_string::operator[]:

```
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
```

The member functions each return a reference to the element of the controlled sequence at position pos. If that position is invalid, the behavior is undefined.

basic_string::push_back:

```
void push_back(value_type c);
```

The member function effectively calls `insert( end(), c)`.

basic_string::rbegin:

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

basic_string::rend:

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member functions each return a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, the function designates the end of the reverse sequence.

basic_string::replace:

```
basic_string& replace(size_type p0, size_type n0,
    const value_type *s);
basic_string& replace(size_type p0, size_type n0,
    const value_type *s, size_type n);
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str);
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str, size_type pos, size_type n);
basic_string& replace(size_type p0, size_type n0,
    size_type n, value_type c);
basic_string& replace(iterator first0, iterator last0,
    const value_type *s);
basic_string& replace(iterator first0, iterator last0,
    const value_type *s, size_type n);
basic_string& replace(iterator first0, iterator last0,
    const basic_string& str);
basic_string& replace(iterator first0, iterator last0,
    size_type n, value_type c);
template<class InIt>
    basic_string&
        replace(iterator first0, iterator last0,
            InIt first, InIt last);
```

The member functions each replace up to n0 elements of the controlled sequence beginning with position p0, or the elements of the controlled sequence beginning with the one pointed to by first, up to but not including last. The replacement is the operand sequence specified by the remaining operands. The function then returns *this. If InIt is an integer type in the template member function, the operand sequence first, last behaves the same as (size_type)first, (value_type)last.

In this implementation, if a translator does not support member template functions, the template:

```
template<class InIt>
    basic_string& replace(iterator first0, iterator last0,
        InIt first, InIt last);
```

is replaced by:

```
basic_string& replace(iterator first0, iterator last0,
    const_pointer first, const_pointer last);
```

basic_string::reserve:

```
void reserve(size_type n = 0);
```

The member function ensures that capacity() henceforth returns at least n.

basic_string::resize:

```
void resize(size_type n, value_type c = value_type());
```

The member function ensures that size() henceforth returns n. If it must make the controlled sequence longer, it appends elements with value c. To make the controlled sequence shorter, the member function effectively calls erase(begin() + n, end()).

basic_string::rfind:

```
size_type rfind(value_type c, size_type pos = npos) const;
size_type rfind(const value_type *s,
    size_type pos = npos) const;
size_type rfind(const value_type *s,
    size_type pos, size_type n = npos) const;
size_type rfind(const basic_string& str,
    size_type pos = npos) const;
```

The member functions each find the last (highest beginning position) subsequence in the controlled sequence, beginning on or before position pos, that matches the operand sequence specified by the remaining operands. If it succeeds, the function returns the position where the matching subsequence begins. Otherwise, it returns npos .

basic_string::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

basic_string::substr:

```
basic_string substr(size_type pos = 0,
    size_type n = npos) const;
```

The member function returns an object whose controlled sequence is a copy of up to n elements of the controlled sequence beginning at position pos.

basic_string::swap:

```
void swap(basic_string& str);
```

The member function swaps the controlled sequences between `*this` and `str`. If `get_allocator() == str.get_allocator()`, it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Operators:

basic_string::operator+=:

```
basic_string& operator+=(value_type c);
basic_string& operator+=(const value_type *s);
basic_string& operator+=(const basic_string& rhs);
```

The operators each append the operand sequence to the end of the sequence controlled by `*this`, then return `*this`.

basic_string::operator=:

```
basic_string& operator=(value_type c);
basic_string& operator=(const value_type *s);
basic_string& operator=(const basic_string& rhs);
```

The operators each replace the sequence controlled by `*this` with the operand sequence, then return `*this`.

char_traits

Description: The template class describes various **character traits** for type `E`. The template class basic_string as well as several iostreams template classes, including basic_ios, use this information to manipulate elements of type `E`. Such an element type must not require explicit construction or destruction. It must supply a default constructor, a copy constructor, and an assignment operator, with the expected semantics. A bitwise copy must have the same effect as an assignment.

Not all parts of the Standard C++ Library rely completely upon the member functions of `char_traits<E>` to manipulate an element. Specifically, formatted input functions and formatted output functions make use of the following additional operations, also with the expected semantics:

- `operator==(E)` and `operator!=(E)` to compare elements
- `(char)ch` to convert an element `ch` to its corresponding single-byte character code, or `'\0'` if no such code exists
- `(E)c` to convert a `char` value `c` to its correponding character code of type `E`

None of the member functions of class `char_traits` may throw exceptions.

Synopsis:

```
template<class E>
    class char_traits {
public:
    typedef E char_type;
    typedef T1 int_type;
    typedef T2 pos_type;
    typedef T3 off_type;
    typedef T4 state_type;
```

```
static void assign(char_type& x, const char_type& y);
static char_type *assign(char_type *x, size_t n,
    char_type y);
static bool eq(const char_type& x,
    const char_type& y);
static bool lt(const char_type& x,
    const char_type& y);
static int compare(const char_type *x,
    const char_type *y, size_t n);
static size_t length(const char_type *x);
static char_type *copy(char_type *x,
    const char_type *y, size_t n);
static char_type *move(char_type *x,
    const char_type *y, size_t n);
static const char_type *find(const char_type *x,
    size_t n, const char_type& y);
static char_type to_char_type(const int_type& ch);
static int_type to_int_type(const char_type& c);
static bool eq_int_type(const int_type& ch1,
    const int_type& ch2);
static int_type eof();
static int_type not_eof(const int_type& ch);
};
```

Types:

char_traits::char_type:

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

char_traits::int_type:

```
typedef T1 int_type;
```

The type is (typically) an integer type `T1` that describes an object that can represent any element of the controlled sequence as well as the value returned by `eof()`. It must be possible to type cast a value of type `E` to `int_type` then back to `E` without altering the original value.

char_traits::off_type:

```
typedef T3 off_type;
```

The type is a signed integer type `T3` that describes an object that can store a byte offset involved in various stream positioning operations. It is typically a synonym for streamoff, but in any case it has essentially the same properties as that type.

char_traits::pos_type:

```
typedef T2 pos_type;
```

The type is an opaque type `T2` that describes an object that can store all the information needed to restore an arbitrary file-position indicator within a stream. It is typically a synonym for streampos, but in any case it has essentially the same properties as that type.

char_traits::state_type:

```
typedef T4 state_type;
```

The type is an opaque type `T4` that describes an object that can represent a conversion state. It is typically a synonym for mbstate_t, but in any case it has essentially the same properties as that type.

Member functions:

char_traits::assign:

```
static void assign(char_type& x, const char_type& y);
static char_type *assign(char_type *x, size_t n,
    char_type y);
```

The first static member function assigns `y` to `x`. The second static member function assigns `y` to each element `X[N]` for `N` in the range `[0, N)`.

char_traits::compare:

```
static int compare(const char_type *x,
    const char_type *y, size_t n);
```

The static member function compares the sequence of length `n` beginning at `x`to the sequence of the same length beginning at `y`. The function returns:

- a negative value if the first differing element in `x` (as determined by eq) compares less than the corresponding element in `y` (as determined by lt)
- zero if the two compare equal element by element
- a positive value otherwise

char_traits::copy:

```
static char_type *copy(char_type *x, const char_type *y,
    size_t n);
```

The static member function copies the sequence of `n` elements beginning at `y` to the array beginning at `x`, then returns `x`. The source and destination must not overlap.

char_traits::eof:

```
static int_type eof();
```

The static member function returns a value that represents end-of-file (such as EOF or WEOF). If the value is also representable as type `E`, it must correspond to no *valid* value of that type.

char_traits::eq:

```
static bool eq(const char_type& x, const char_type& y);
```

The static member function returns true if `x` compares equal to `y`.

char_traits::eq_int_type:

```
static bool eq_int_type(const int_type& ch1,
    const int_type& ch2);
```

The static member function returns true if `ch1 == ch2`.

char_traits::find:

```
static const char_type *find(const char_type *x,
    size_t n, const char_type& y);
```

The static member function determines the lowest N in the range [0, n) for which eq(x[N], y) is true. If successful, it returns x + N. Otherwise, it returns a null pointer.

char_traits::length:

```
static size_t length(const char_type *x);
```

The static member function returns the number of elements N in the sequence beginning at x up to but not including the element x[N] which compares equal to char_type().

char_traits::lt:

```
static bool lt(const char_type& x, const char_type& y);
```

The static member function returns true if x compares less than y.

char_traits::move:

```
static char_type *move(char_type *x, const char_type *y,
    size_t n);
```

The static member function copies the sequence of n elements beginning at y to the array beginning at x, then returns x. The source and destination may overlap.

char_traits::not_eof:

```
static int_type not_eof(const int_type& ch);
```

If !eq_int_type(eof(), ch), the static member function returns ch. Otherwise, it returns a value other than eof().

char_traits::to_char_type:

```
static char_type to_char_type(const int_type& ch);
```

The static member function returns ch, represented as type E. A value of ch that cannot be so represented yields an unspecified result.

char_traits::to_int_type:

```
static int_type to_int_type(const char_type& c);
```

The static member function returns ch, represented as type int_type. It should always be true that to_char_type(to_int_type(c) == c for any value of c.

char_traits<char>

```
template<>
    class char_traits<char>;
```

The class is an explicit specialization of template class char_traits for elements of type *char,* (so that it can take advantage of library functions that manipulate objects of this type).

char_traits<wchar_t>

```
template<>
    class char_traits<wchar_t>;
```

The class is an explicit specialization of template class char_traits for elements of type wchar_t (so that it can take advantage of library functions that manipulate objects of this type).

Template functions

getline

```
template<class E, class T, class A>
    basic_istream<E, T>& getline(
        basic_istream<E, T>& is,
        basic_string<E, T, A>& str);
template<class E, class T, class A>
    basic_istream<E, T>& getline(
        basic_istream<E, T>& is,
        basic_string<E, T, A>& str,
        E delim);
```

The first function returns `getline(is, str, is.widen('\n'))`.

The second function replaces the sequence controlled by `str` with a sequence of elements extracted from the stream `is`. In order of testing, extraction stops:

1. at end of file
2. after the function extracts an element that compares equal to `delim`, in which case the element is neither put back nor appended to the controlled sequence
3. after the function extracts `str.max_size()` elements, in which case the function calls `setstate(ios_base::failbit)`.

If the function extracts no elements, it calls `setstate(failbit)`. In any case, it returns `*this`.

operator+

```
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        E rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        E lhs,
        const basic_string<E, T, A>& rhs);
```

The functions each overload `operator+` to concatenate two objects of template class basic_string. All effectively return `basic_string<E, T, A>(lhs).append(rhs)`.

operator!=

```
template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
```

```
    bool operator!=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overload `operator!=` to compare two objects of template class basic_string. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) != 0`.

operator==

```
template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator==(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overload `operator==` to compare two objects of template class basic_string. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) == 0`.

operator<

```
template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator<(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overload `operator<` to compare two objects of template class basic_string. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) < 0`.

operator<<

```
template<class E, class T, class A>
    basic_ostream<E, T>& operator<<(
        basic_ostream<E, T>& os,
        const basic_string<E, T, A>& str);
```

The template function overloads `operator<<` to insert an object `str` of template class basic_string into the stream `os` The function effectively returns `os.write( str.c_str(), str.size())`.

operator<=

```
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
```

```
        const E *rhs);
template<class E, class T, class A>
    bool operator<=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overload `operator<=` to compare two objects of template class basic_string. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) <= 0`.

operator>

```
template<class E, class T, class A>
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overload `operator>` to compare two objects of template class basic_string. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) > 0`.

operator>=

```
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overload `operator>=` to compare two objects of template class basic_string. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) >= 0`.

operator>>

```
template<class E, class T, class A>
    basic_istream<E, T>& operator>>(
        basic_istream<E, T>& is,
        const basic_string<E, T, A>& str);
```

The template function overloads `operator>>` to replace the sequence controlled by `str` with a sequence of elements extracted from the stream `is`. Extraction stops:

- at end of file
- after the function extracts `is.width()` elements, if that value is nonzero
- after the function extracts `is.max_size()` elements
- after the function extracts an element `c` for which `use_facet< ctype<E> >( getloc()). is( ctype<E>::space, c)` is true, in which case the character is put back

If the function extracts no elements, it calls setstate(ios_base::failbit). In any case, it calls is.width(0) and returns *this.

swap

```
template<class T, class A>
    void swap(
        basic_string<E, T, A>& lhs,
        basic_string<E, T, A>& rhs);
```

The template function executes lhs.swap(rhs).

Types

string

```
typedef basic_string<char> string;
```

The type describes a specialization of template class basic_string specialized for elements of type *char.*

wstring

```
typedef basic_string<wchar_t> wstring;
```

The type describes a specialization of template class basic_string for elements of type wchar_t.

<strstream>

Description

Include the iostreams standard header **<strstream>** to define several classes that support iostreams operations on sequences stored in an allocated array of *char* object. Such sequences are easily converted to and from C strings.

Synopsis

```
namespace std {
class strstreambuf;
class istrstream;
class ostrstream;
class strstream;
    }
```

Classes

strstreambuf

Description: The class describes a **stream buffer** that controls the transmission of elements to and from a sequence of elements stored in a *char* array object. Depending on how it is constructed, the object can be allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class strstreambuf stores several bits of mode information as its **strstreambuf mode**. These bits indicate whether the controlled sequence:

- has been **allocated**, and hence needs to be freed eventually
- is **modifiable**
- is **extendable** by reallocating storage

- has been **frozen** and hence needs to be unfrozen before the object is destroyed, or freed (if allocated) by an agency other than the object

A controlled sequence that is frozen cannot be modified or extended, regardless of the state of these separate mode bits.

The object also stores pointers to two functions that control **strstreambuf allocation**. If these are null pointers, the object devises its own method of allocating and freeing storage for the controlled sequence.

Synopsis:

```
class strstreambuf : public streambuf {
public:
    explicit strstreambuf(streamsize n = 0);
    strstreambuf(void (*palloc)(size_t),
        void (*pfree)(void *));
    strstreambuf(char *gp, streamsize n,
        char *pp = 0);
    strstreambuf(signed char *gp, streamsize n,
        signed char *pp = 0);
    strstreambuf(unsigned char *gp, streamsize n,
        unsigned char *pp = 0);
    strstreambuf(const char *gp, streamsize n);
    strstreambuf(const signed char *gp, streamsize n);
    strstreambuf(const unsigned char *gp, streamsize n);
    void freeze(bool frz = true);
    char *str();
    streamsize pcount();
protected:
    virtual streampos seekoff(streamoff off,
        ios_base::seekdir way,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    virtual streampos seekpos(streampos sp,
        ios_base::openmode which =
            ios_base::in | ios_base::out);
    virtual int underflow();
    virtual int pbackfail(int c = EOF);
    virtual int overflow(int c = EOF);
    };
```

Constructor:

strstreambuf::strstreambuf:

```
explicit strstreambuf(streamsize n = 0);
strstreambuf(void (*palloc)(size_t),
    void (*pfree)(void *));
strstreambuf(char *gp, streamsize n,
    char *pp = 0);
strstreambuf(signed char *gp, streamsize n,
    signed char *pp = 0);
strstreambuf(unsigned char *gp, streamsize n,
    unsigned char *pp = 0);
strstreambuf(const char *gp, streamsize n);
strstreambuf(const signed char *gp, streamsize n);
strstreambuf(const unsigned char *gp, streamsize n);
```

The first constructor stores a null pointer in all the pointers controlling the input buffer, the output buffer, and strstreambuf allocation. It sets the stored strstreambuf mode to make the controlled sequence modifiable and extendable.

The second constructor behaves much as the first, except that it stores `palloc` as the pointer to the function to call to allocate storage, and `pfree` as the pointer to the function to call to free that storage.

The three constructors:

```
strstreambuf(char *gp, streamsize n,
    char *pp = 0);
strstreambuf(signed char *gp, streamsize n,
    signed char *pp = 0);
strstreambuf(unsigned char *gp, streamsize n,
    unsigned char *pp = 0);
```

also behave much as the first, except that `gp` designates the array object used to hold the controlled sequence. (Hence, it must not be a null pointer.) The number of elements `N` in the array is determined as follows:

- If `(n > 0)`, then `N` is `n`.
- If `(n == 0)`, then `N` is `strlen((const char *)gp)`.
- If `(n < 0)`, then `N` is `INT_MAX`.

If `pp` is a null pointer, the function establishes just an input buffer, by executing:

```
setg(gp, gp, gp + N);
```

Otherwise, it establishes both input and output buffers, by executing:

```
setg(gp, gp, pp);
setp(pp, gp + N);
```

In this case, `pp` must be in the interval `[gp, gp + N]`.

Finally, the three constructors:

```
strstreambuf(const char *gp, streamsize n);
strstreambuf(const signed char *gp, streamsize n);
strstreambuf(const unsigned char *gp, streamsize n);
```

all behave the same as:

```
streambuf((char *)gp, n);
```

except that the stored mode makes the controlled sequence neither modifiable not extendable.

Member functions:

strstreambuf::freeze:

```
void freeze(bool frz = true);
```

If `frz` is true, the function alters the stored strstreambuf mode to make the controlled sequence frozen. Otherwise, it makes the controlled sequence not frozen.

strstreambuf::pcount:

```
streamsize pcount();
```

The member function returns a count of the number of elements written to the controlled sequence. Specifically, if `pptr()` is a null pointer, the function returns zero. Otherwise, it returns `pptr() - pbase()`.

strstreambuf::overflow:

```
virtual int overflow(int c = EOF);
```

If `c != EOF`, the protected virtual member function endeavors to insert the element `(char)c` into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- If the stored strstreambuf mode says the controlled sequence is modifiable, extendable, and not frozen, the function can make a write position available by allocating new for the output buffer. (Extending the output buffer this way also extends any associated input buffer.)

If the function cannot succeed, it returns `EOF`. Otherwise, if `c == EOF` it returns some value other than `EOF`. Otherwise, it returns `c`.

strstreambuf::pbackfail:

```
virtual int pbackfail(int c = EOF);
```

The protected virtual member function endeavors to put back an element into the input buffer, then make it the current element (pointed to by the next pointer).

If `c = EOF`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `x = (char)c`. The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to `x`, it can simply decrement the next pointer for the input buffer.
- If a putback position is available, and if the strstreambuf mode says the controlled sequence is modifiable, the function can store `x` into the putback position and decrement the next pointer for the input buffer.

If the function cannot succeed, it returns `EOF`. Otherwise, if `c == EOF` it returns some value other than `EOF`. Otherwise, it returns `c`.

strstreambuf::seekoff:

```
virtual streampos seekoff(streamoff off,
    ios_base::seekdir way,
    ios_base::openmode which =
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `strstreambuf`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If `way == ios_base::beg`, the new position is the beginning of the stream plus `off`.
- If `way == ios_base::cur`, the new position is the current stream position plus `off`.
- If `way == ios_base::end`, the new position is the end of the stream plus `off`.

If `which & ios_base::in` is nonzero and the input buffer exist, the function alters the next position to read in the input buffer. If `which & ios_base::out` is also nonzero, `way != ios_base::cur`, and the output buffer exists, the function also sets the next position to write to match the next position to read.

Otherwise, if `which & ios_base::out` is nonzero and the output buffer exists, the function alters the next position to write in the output buffer. Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

If the function succeeds in altering the stream position(s), it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

strstreambuf::seekpos:

```
virtual streampos seekpos(streampos sp,
    ios_base::openmode which =
        ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `strstreambuf`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by `sp`.

If `which & ios_base::in` is nonzero and the input buffer exists, the function alters the next position to read in the input buffer. (If `which & ios_base::out` is nonzero and the output buffer exists, the function also sets the next position to write to match the next position to read.) Otherwise, if `which & ios_base::out` is nonzero and the output buffer exists, the function alters the next position to write in the output buffer. Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

If the function succeeds in altering the stream position(s), it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

strstreambuf::str:

```
char *str();
```

The member function calls freeze(), then returns a pointer to the beginning of the controlled sequence. (Note that no terminating null element exists, unless you insert one explicitly.)

strstreambuf::underflow:

```
virtual int underflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input buffer, then advance the current stream position, and return the element as `(int)(unsigned char)c`. It can do so in only one way: If a read position is available, it takes `c` as the element stored in the read position and advances the next pointer for the input buffer.

If the function cannot succeed, it returns EOF. Otherwise, it returns the current element in the input stream, converted as described above.

istrstream

Description: The class describes an object that controls extraction of elements and encoded objects from a stream buffer of class strstreambuf. The object stores an ojbect of class `strstreambuf`.

Synopsis:

```
class istrstream : public istream {
public:
    explicit istrstream(const char *s);
```

```
    explicit istrstream(char *s);
    istrstream(const char *s, streamsize n);
    istrstream(char *s, streamsize n);
    strstreambuf *rdbuf() const;
    char *str();
    };
```

Constructor:

istrstream::istrstream:

```
explicit istrstream(const char *s);
explicit istrstream(char *s);
istrstream(const char *s, streamsize n);
istrstream(char *s, streamsize n);
```

All the constructors initialize the base class by calling `istream(sb)`, where `sb` is the stored object of class `strstreambuf`. The first two constructors also initialize `sb` by calling `strstreambuf((const char *)s, 0)`. The remaining two constructors instead call `strstreambuf((const char *)s, n)`.

Member functions:

istrstream::rdbuf:

```
strstreambuf *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to strstreambuf.

istrstream::str:

```
char *str();
```

The member function returns `rdbuf()-> str()`.

ostrstream

Description: The class describes an object that controls insertion of elements and encoded objects into a stream buffer of class `strstreambuf`. The object stores an ojbect of class `strstreambuf`.

Synopsis:

```
class ostrstream : public ostream {
public:
    ostrstream();
    ostrstream(char *s, streamsize n,
        ios_base::openmode mode = ios_base::out);
    strstreambuf *rdbuf() const;
    void freeze(bool frz = true);
    char *str();
    streamsize pcount() const;
    };
```

Constructor:

ostrstream::ostrstream:

```
ostrstream();
ostrstream(char *s, streamsize n,
    ios_base::openmode mode = ios_base::out);
```

Both constructors initialize the base class by calling `ostream(sb)`, where `sb` is the stored object of class `strstreambuf`. The first constructor also initializes `sb` by calling `strstreambuf()`. The second constructor initializes the base class one of two ways:

- If `mode & ios_base::app == 0`, then `s` must designate the first element of an array of `n` elements, and the constructor calls `strstreambuf(s, n, s)`.
- Otherwise, `s` must designate the first element of an array of `n` elements that contains a C string whose first element is designated by `s`, and the constructor calls `strstreambuf(s, n, s + strlen(s)`.

Member functions:

ostrstream::freeze:

```
void freeze(bool frz = true)
```

The member function calls `rdbuf()-> freeze(frz)`.

ostrstream::pcount:

```
streamsize pcount() const;
```

The member function returns `rdbuf()-> pcount()`.

ostrstream::rdbuf:

```
strstreambuf *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to strstreambuf.

ostrstream::str:

```
char *str();
```

The member function returns `rdbuf()-> str()`.

strstream

Description: The class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class strstreambuf. The object stores an ojbect of class `strstreambuf`.

Synopsis:

```
class strstream : public iostream {
public:
    strstream();
    strstream(char *s, streamsize n,
        ios_base::openmode mode =
            ios_base::in | ios_base::out);
    strstreambuf *rdbuf() const;
    void freeze(bool frz = true);
    char *str();
    streamsize pcount() const;
    };
```

Constructor:

strstream::strstream:

```
strstream();
strstream(char *s, streamsize n,
    ios_base::openmode mode =
        ios_base::in | ios_base::out);
```

Both constructors initialize the base class by calling `streambuf(sb)`, where `sb` is the stored object of class strstreambuf. The first constructor also initializes `sb` by calling `strstreambuf()`. The second constructor initializes the base class one of two ways:

- If `mode & ios_base::app == 0`, then `s` must designate the first element of an array of `n` elements, and the constructor calls `strstreambuf(s, n, s)`.
- Otherwise, `s` must designate the first element of an array of `n` elements that contains a C string whose first element is designated by `s`, and the constructor calls `strstreambuf(s, n, s + strlen(s)`.

Member functions:

strstream::freeze:

```
void freeze(bool frz = true)
```

The member function calls `rdbuf()-> freeze(zfrz)`.

strstream::pcount:

```
streamsize pcount() const;
```

The member function returns `rdbuf()-> pcount()`.

strstream::rdbuf:

```
strstreambuf *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to strstreambuf.

strstream::str:

```
char *str();
```

The member function returns `rdbuf()-> str()`.

<tuple>

Description

Include the TR1 header **<tuple>** to define a template `tuple` whose instances hold objects of varying types.

Note: With the next revision of the C++ Standard (C++0X), this header makes extensive use of **variadic templates**, indicated by various uses of ellipsis (...). The descriptions below use C++0X notation, but still apply to older compilers, provided the number of parameters in a varying-length list does not exceed ten.

Note: To enable this header file, you must define the macro __IBMCPP_TR1__ and use the **TARGET** compiler option to specify a valid release level. Valid release levels for TR1 support are zOSV1R12 or later. For example, you can specify the **TARGET** option as follows: `TARGET(zOSV1R12)`.

Any release prior to zOSV1R12 is invalid for the use of TR1. If TR1 code is used in an application to be compiled on an earlier platform, the compiler will issue the `#error` directive.

Synopsis

```
namespace std {
    namespace tr1 {

    // TEMPLATE CLASSES
 template<class... Types>
    class tuple;

 template<int Idx, class Ty>
     class tuple_element; // not defined
 template<int Idx, class... Types>
     class tuple_element<Idx, tuple<Types...>>;

template<class Ty>
     class tuple_size; // not defined
 template<class... Types>
     class tuple_size<tuple<Types...>>;

    // TEMPLATE Functions
 template<int Idx, class Types>
     typename tuple_element <Idx, tuple<Types...>>::type&
         get(tuple<Types...&> tpl);
 template<int Idx, class Types>
     typename tuple_element <Idx, tuple<Types...>>::type&
         get(const tuple<Types...&> tpl);

 template<class... Types>
     tuple<Types2...>
         make_tuple(Types...);
 template<class... Types>
     tuple<Types&...>
         Tie(Types&...);

    // TEMPLATE COMPARISON OPERATORS
 template<class... Types1, class... Types2>
    bool operator==(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;
 template<class... Types1, class... Types2>
    bool operator!=(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;
 template<class... Types1, class... Types2>
    bool operator<(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;
 template<class... Types1, class... Types2>
    bool operator<=(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;
 template<class... Types1, class... Types2>
    bool operator>(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;
 template<class... Types1, class... Types2>
    bool operator>=(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;

    //CONST OBJECTS
const T1 ignore;

    }  // namespace tr1
}  // namespace std
```

Classes

tuple

Description: The template class describes an object that stores zero or more objects of types specified by `Types`. The **extent** of a tuple instance is the number `N` of its template arguments. The **index** of the template argument `Ti` (counting from zero) and of the corresponding stored value of that type is `i`.

Synopsis:

```
template<class... Types>
class tuple {
public:
    tuple();
    explicit tuple(const Types&...);
    tuple(const tuple& right);
    template <class... Types2>
        tuple(const tuple<Types2...>& right);
    template <class U1, class U2>
        tuple(const pair<U1, U2>& right);

    tuple(tuple&& right)
    template <class... Types2>
        tuple(tuple<Types2...>&& right);
    template <class U1, class U2>
        tuple(pair<U1, U2>&& right);

    tuple& operator=(const tuple& right);
    template <class... Types2>
        tuple& operator=(const tuple<Types2...>& right);
    template <class U1, class U2>
        tuple& operator=(const pair<U1, U2>& right);

    tuple& operator=(tuple&& right);
    template <class... Types2>
        tuple& operator=(tuple<Types2...>&& right);
    template <class U1, class U2>
        tuple& operator=(pair<U1, U2>&& right);
    };
```

Constructor:

tuple::tuple:

```
tuple();
    explicit tuple(const Types&...);
    tuple(const tuple& right);
    template <class... Types2>
        tuple(const tuple<Types2...>& right);
    template <class U1, class U2>
        tuple(const pair<U1, U2>& right);

tuple(tuple&& right)
    template <class... Types2>
        tuple(tuple<Types2...>&& right);
    template <class U1, class U2>
        tuple(pair<U1, U2>&& right);
```

The first constructor constructs an object whose elements are default constructed. The second constructor constructs an object whose elements are copy constructed from the argument list. The third and fourth constructors construct an object whose elements are copy constructed from the corresponding element of `right`. The fifth constructor constructs an object whose element at index 0 is copy constructed from `right.first` and whose element at index 1 is copy constructed from `right.second`.

Operators:

tuple::operator=:

```
tuple& operator=(const tuple& right);
    template <class... Types2>
        tuple& operator=(const tuple<Types2...>& right);
    template <class U1, class U2>
        tuple& operator=(const pair<U1, U2>& right);

    tuple& operator=(tuple&& right);
    template <class... Types2>
        tuple& operator=(tuple<Types2...>&& right);
    template <class U1, class U2>
        tuple& operator=(pair<U1, U2>&& right);
```

The first two member operators assign the elements of `right` to the corresponding elements of `*this`. The third member operator assigns `right.first` to the element at index 0 of `*this` and `right.second` to the element at index 1. All three member operators return `*this`.

tuple_element

```
template<int Idx, class Ty>
    class tuple_element; // not defined
template<int Idx, class... Types>
    class tuple_element<Idx, tuple<Types...>>;
```

The template class has a member type `Type` that is a synonym for the type at index `Idx` of the type `Tuple<Types...>`.

tuple_size

```
template<class Ty>
    class tuple_size; // not defined
template<class... Types>
    class tuple_size<tuple<Types...>>;
```

The template class has a member `const int value` whose value is the extent of the type `Tuple<Types...>`.

Functions

get

```
template<int Idx, class... Types>
    typename tuple_element <Idx, tuple<Types...>>::type&
        get(tuple<Types...&> tpl);
 template<int Idx, class Types>
     typename tuple_element <Idx, tuple<Types...>>::type&
        get(const tuple<Types...&> tpl);
```

The template functions return a reference to the value at index `Idx` in the `tuple` object `tpl`. If the corresponding type `Ti` in `Types` is a reference type, both functions return `Ti`; otherwise the first function returns `Ti&` and the second function returns `const Ti&`.

make_tuple

```
template<class... Types>
    tuple<Types2...>
    make_tuple(Types...);
```

The template function returns a `tuple` object constructed from the argument list, where each type `T2i` in `Types2` is the same as `Ti` in `Types`, except where `Ti` is `reference_wrapper<X>`, in which case `T2i` is `X`.

operator==

```
template<class... Types1, class... Types2>
    bool operator==(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;
```

The function returns true only when both tuples are empty, or when `get<0>(tpl1) == get<0>(tpl2) &&...` for all corresponding elements.

operator!=

```
template<class... Types1, class... Types2>
    bool operator!=(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;
```

The function returns `!(tpl1 == tpl2)`.

operator<

```
template<class... Types1, class... Types2>
    bool operator<(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;
```

The function returns true only when both tuples are not empty and `get<0>(tpl1) < get<0>(tpl2) || !(get<0>(tpl2) < get<0>(tpl1)) &&...` for all corresponding elements.

operator<=

```
template<class... Types1, class... Types2>
    bool operator<=(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;
```

The function returns `!(tpl2 < tpl1)`.

operator>

```
template<class... Types1, class... Types2>
    bool operator>(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;
```

The function returns `tpl2 < tpl1`.

operator>=

```
template<class... Types1, class... Types2>
    bool operator>=(const tuple<Types1...>& tpl1,
    const tuple<Types2...>& tpl2;
```

The function returns `!(tpl1 < tpl2)`.

tie

```
template<class... Types>
     tuple<Types&...>
         Tie(Types&...);
```

The template function returns a `tuple` object constructed from the argument list, where each element is a reference. Note that a reference to **ignore** can be assigned anything and will do nothing.

<typeinfo>

Description

Include the standard header **<typeinfo>** to define several types associated with the type-identification operator **typeid**, which yields information about both static and dynamic types.

Synopsis

```
namespace std {
class type_info;
class bad_cast;
class bad_typeid;
    }
```

Classes

bad_cast

```
class bad_cast : public exception {
    };
```

The class describes an exception thrown to indicate that a **dynamic cast** expression, of the form:

```
dynamic_cast<type>(expression)
```

generated a null pointer to initialize a reference. The value returned by `what()` is an implementation-defined C string. None of the member functions throw any exceptions.

bad_typeid

```
class bad_typeid : public exception {
    };
```

The class describes an exception thrown to indicate that a typeid operator encountered a null pointer. The value returned by `what()` is an implementation-defined C string. None of the member functions throw any exceptions.

type_info

Description: The class describes type information generated within the program by the implementation. Objects of this class effectively store a pointer to a **name** for the type, and an encoded value suitable for comparing two types for equality or **collating order**. The names, encoded values, and collating order for types are all unspecified and may differ between program executions.

An expression of the form `typeid x` is the *only* way to construct a (temporary) `typeinfo` object. The class has only a private copy constructor. Since the assignment operator is also private, you cannot copy or assign objects of class `typeinfo` either.

Synopsis:

```
class type_info {
public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
```

```
    bool before(const type_info& rhs) const;
    const char *name() const;
private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
    };
```

Member functions:

type_info::operator!=:

```
bool operator!=(const type_info& rhs) const;
```

The function returns `!(*this == rhs)`.

type_info::operator==:

```
bool operator==(const type_info& rhs) const;
```

The function returns a nonzero value if `*this` and `rhs` represent the same type.

type_info::before:

```
bool before(const type_info& rhs) const;
```

The function returns a nonzero value if `*this` precedes `rhs` in the `collating order` for types.

type_info::name:

```
const char *name() const;
```

The function returns a C string which specifies the `name` of the type.

<type_traits>

Description

Include the TR1 header **<type_traits>** to define several templates that provide compile-time constants giving information about the properties of their type arguments.

Note: To enable this header file, you must define the macro __IBMCPP_TR1__ and use the **TARGET** compiler option to specify a valid release level. Valid release levels for TR1 support are zOSV1R12 or later. For example, you can specify the **TARGET** option as follows: `TARGET(zOSV1R12)`.

Any release prior to zOSV1R12 is invalid for the use of TR1. If TR1 code is used in an application to be compiled on an earlier platform, the compiler will issue the `#error` directive.

Synopsis

```
namespace std {
    namespace tr1 {
    // HELPER CLASSES
template <class Ty, Ty v>
   struct integral_constant;
typedef integral_constant<bool, false> false_type;
typedef integral_constant<bool, true> true_type;

    // TYPE CATEGORIES
```

```
template <class Ty> struct is_void;
template <class Ty> struct is_integral;
template <class Ty> struct is_floating_point;
template <class Ty> struct is_array;
template <class Ty> struct is_pointer;
template <class Ty> struct is_reference;
template <class Ty> struct is_member_object_pointer;
template <class Ty> struct is_member_function_pointer;
template <class Ty> struct is_enum;
template <class Ty> struct is_union;
template <class Ty> struct is_class;
template <class Ty> struct is_function;
template <class Ty> struct is_arithmetic;
template <class Ty> struct is_fundamental;
template <class Ty> struct is_object;
template <class Ty> struct is_scalar;
template <class Ty> struct is_compound;
template <class Ty> struct is_member_pointer;

    // TYPE PROPERTIES
template <class Ty> struct is_const;
template <class Ty> struct is_volatile;
template <class Ty> struct is_pod;
template <class Ty> struct is_empty;
template <class Ty> struct is_polymorphic;
template <class Ty> struct is_abstract;
template <class Ty> struct has_trivial_constructor;
template <class Ty> struct has_trivial_copy;
template <class Ty> struct has_trivial_assign;
template <class Ty> struct has_trivial_destructor;
template <class Ty> struct has_nothrow_constructor;
template <class Ty> struct has_nothrow_copy;
template <class Ty> struct has_nothrow_assign;
template <class Ty> struct has_virtual_destructor;
template <class Ty> struct is_signed;
template <class Ty> struct is_unsigned;
template <class Ty> struct alignment_of;
template <class Ty> struct rank;
template <class Ty, unsigned _I = 0> struct extent;

    // TYPE COMPARISONS
template <class Ty1, class Ty2> struct is_same;
template <class From, class To> struct is_convertible;
template <class Base, class Derived> struct is_base_of;

    // CONST-VOLATILE MODIFICATIONS
template <class Ty> struct remove_const;
template <class Ty> struct remove_volatile;
template <class Ty> struct remove_cv;
template <class Ty> struct add_const;
template <class Ty> struct add_volatile;
template <class Ty> struct add_cv;

    // REFERENCE MODIFICATIONS
template <class Ty> struct remove_reference;
template <class Ty> struct add_reference;

    // ARRAY MODIFICATIONS
template <class Ty> struct remove_extent;
template <class Ty> struct remove_all_extents;

    // POINTER MODIFICATIONS
template <class Ty> struct remove_pointer;
template <class Ty> struct add_pointer;
```

```
    //OTHER MODIFICATIONS
template <class Ty> struct aligned_storage;
    }  // namespace tr1
}  // namespace std
```

Implementation Notes

1. The circumstances under which these type traits yield a result of "true" is not specified in TR1:
   - is_empty
   - is_pod
   - has_trivial_constructor
   - has_trivial_copy
   - has_trivial_assign
   - has_trivial_destructor
   - has_nothrow_destructor
   - has_nothrow_copy
   - has_nothrow_assign

   With XL C++, these traits behave as specified in the following section.
2. TR1 grants to implementors of the type traits library the latitude to implement certain type traits as class templates with no static or non-static data or function members, no base classes, and no nested types. For example, the following implementations of the type traits `is_class` and `is_union` are permissible for implementations that cannot distinguish between class and union types:

   ```
   template <typename T> struct is_class{};
   template <typename T> struct is_union{};
   ```

   The type traits for which this latitude is granted are:
   - is_class
   - is_union
   - is_polymorphic
   - is_abstract

   XL C++ does not take advantage of this latitude. Full implementations of these type traits are provided
3. TR1 grants to implementors of the type traits library the latitude to implement the type trait has_virtual_destructor in such a way that its static data member always has a value of `true`, regardless of the type argument to which it is applied. XL C++ does not take advantage of this latitude. The expression `has_virtual_destructor<T>::value` will have a value of `true` if and only if the type argument `T` is a class type with a virtual destructor.

Helper Class

Description

The template class, when specialized with an integral type and a value of that type, represents an object that holds a constant of that integral type with the specified value.

Synopsis

```
template <class Ty, Ty v>
 struct integral_constant {
    static const Ty value = v;
    typedef Ty value_type;
```

```
    typedef integral_constant<Ty, v> type;
    };
 typedef integral_constant<bool, true> true_type;
 typedef integral_constant<bool, false> false_type;
```

Types

false_type:

```
typedef integral_constant<bool, false> false_type;
```

The type is a synonym for a specialization of the template `integral_constant`.

true_type:

```
typedef integral_constant<bool, true> true_type;
```

The type is a synonym for a specialization of the template `integral_constant`.

Unary Type Traits

Unary Type Traits describe a property of a single type. Every Unary Type Trait possesses a static data member named `value`. For most traits, this member has type `bool`, and indicates the presence or absence of a specific property or trait of the argument type. For example, the value of the following expression will be true if the type argument `Ty` is a union type, and false otherwise:

```
std::tr1::is_union<Ty>::value
```

A few of the Unary Type Traits possess a static data member named `value` whose type is not `bool`. An example is the type trait `extent`, which gives the number of elements in an array type:

```
typedef char arr[42];
size_t sz = std::tr1::extent<arr>::value; //sz == 42;
```

Every instance of a Unary Type Trait is derived from an instance of `integral_constant`. All Unary Type Traits are default-constructible.

Primary Type Categories

A given type `Ty` will satisfy one of the following categories.

is_void:

```
template <class Ty>
    struct is_void;
```

An instance of the type predicate holds true if the type `Ty` is `void` or a `cv-qualified` form of `void`, otherwise it holds false.

is_integral:

```
template <class Ty>
    struct is_integral;
```

An instance of the type predicate holds true if the type `Ty` is one of the integral types, or a `cv-qualified` form of one of the integral types, otherwise it holds false.

An integral type is one of `bool`, `char`, `unsigned char`, `signed char`, `wchar_t`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, and `unsigned long`. In addition, with compilers that provide them, an integral type can be one of `long long`, `unsigned long long`, `__int64`, and `unsigned __int64`.

is_floating_point:

```
template <class Ty>
    struct is_floating_point;
```

An instance of the type predicate holds true if the type `Ty` is a floating-point type or a `cv-qualified` form of a floating-point type, otherwise it holds false.

A floating-point type is one of `float`, `double`, or `long double`.

is_array:

```
template <class Ty>
    struct is_array;
```

An instance of the type predicate holds true if the type `Ty` is an array type, otherwise it holds false.

is_pointer:

```
template <class Ty>
    struct is_pointer;
```

An instance of the type predicate holds true if the type `Ty` is a pointer to `void`, a pointer to an object, or a pointer to a function, or a `cv-qualified` form of one of them, otherwise it holds false. Note that `is_pointer` holds false if `Ty` is a pointer to member or a pointer to member function.

is_reference:

```
template <class Ty>
    struct is_reference;
```

An instance of the type predicate holds true if the type `Ty` is a reference to an object or to a function, otherwise it holds false.

is_member_object_pointer:

```
template <class Ty>
    struct is_member_object_pointer;
```

An instance of the type predicate holds true if the type `Ty` is a pointer to member object or a `cv-qualified` pointer to member object, otherwise it holds false. Note that `is_member_object_pointer` holds false if `Ty` is a pointer to member function.

is_member_function_pointer:

```
template <class Ty>
    struct is_member_function_pointer;
```

An instance of the type predicate holds true if the type `Ty` is a pointer to member function or a `cv-qualified` pointer to member function, otherwise it holds false.

is_enum:

```
template <class Ty>
     struct is_enum;
```

An instance of the type predicate holds true if the type `Ty` is an enumeration type or a `cv-qualified` form of an enumeration type, otherwise it holds false.

is_union:

```
template <class Ty>
    struct is_union;
```

An instance of the type predicate holds true if the type Ty is a union type or a cv-qualified form of a union type, otherwise it holds false.

is_class:

```
template <class Ty>
    struct is_class;
```

An instance of the type predicate holds true if the type Ty is a type defined as a class or a struct, or a cv-qualified form of one of them, otherwise it holds false.

is_function:

```
template <class Ty>
    struct is_function;
```

An instance of the type predicate holds true if the type Ty is a function type, otherwise it holds false.

Composite Type Traits

is_arithmetic:

```
template <class Ty>
    struct is_arithmetic;
```

An instance of the type predicate holds true if the type Ty is an arithmetic type, that is, an integral type or a floating-point type, or a cv-qualified form of one of them, otherwise it holds false.

is_fundamental:

```
template <class Ty>
    struct is_fundamental;
```

An instance of the type predicate holds true if the type Ty is a fundamental type, that is, void, an integral type, a floating-point type, or a cv-qualified form of one of them, otherwise it holds false.

is_object:

```
template <class Ty>
    struct is_object;
```

An instance of the type predicate holds false if the type Ty is a reference type, a function type, or void, or a cv-qualified form of one of them, otherwise holds true.

is_scalar:

```
template <class Ty>
    struct is_scalar;
```

An instance of the type predicate holds true if the type Ty is an integral type, a floating-point type, an enumeration type, a pointer type, or a pointer to member type, or a cv-qualified form of one of them, otherwise it holds false.

is_compound:

```
template <class Ty>
    struct is_compound;
```

An instance of the type predicate holds false if the type Ty is a scalar type (that is, if is_scalar<Ty> holds true), otherwise it holds true. Thus, the predicate holds true if Ty is an array type, a function type, a pointer to void or an object or a function, a reference, a class, a union, an enumeration, or a pointer to non-static class member, or a cv-qualified form of one of them.

is_member_pointer:

```
template <class Ty>
    struct is_member_pointer;
```

An instance of the type predicate holds true if the type Ty is a pointer to member function or a pointer to member object, or a cv-qualified form of one of them, otherwise it holds false.

Type Properties

is_const:

```
template <class Ty>
    struct is_const;
```

An instance of the type predicate holds true if Ty is const-qualified.

is_volatile:

```
template <class Ty>
    struct is_volatile;
```

An instance of the type predicate holds true if Ty is volatile-qualified.

is_pod:

```
template <class Ty>
    struct is_pod;
```

An instance of the type predicate holds true if the type Ty is a scalar type, a POD aggregate type, or a cv-qualified form of one of them, or an array of such a type, otherwise it holds false.

A **POD aggregate** type is a class, struct, or union whose non-static data members are all scalar types or POD aggregates, and that has no references, no user-defined copy assignment operator, and no user-defined destructor.

is_empty:

```
template <class Ty>
    struct is_empty;
```

An instance of the type predicate holds true if the type Ty is an empty class, otherwise it holds false.

is_polymorphic:

```
template <class Ty>
    struct is_polymorphic;
```

An instance of the type predicate holds true if the type Ty is a class that declares or inherits a virtual function, otherwise it holds false.

is_abstract:

```
template <class Ty>
    struct is_abstract;
```

An instance of the type predicate holds true if the type Ty is a class that has at least one pure virtual function, otherwise it holds false.

has_trivial_constructor:

```
template <class Ty>
    struct has_trivial_constructor;
```

An instance of the type predicate holds true if the type Ty is a class that has a trivial constructor, otherwise it holds false.

A constructor for a class Ty is **trivial** if:
- it is an implicitly declared default constructor
- the class Ty has no virtual functions
- the class Ty has no virtual bases
- all the direct bases of the class Ty have trivial constructors
- the classes of all the non-static data members of class type have trivial constructors
- the classes of all the non-static data members of type array of class have trivial constructors

has_trivial_assign:

```
template <class _T> struct has_trivial_assign;
```

An instance of the type predicate holds true if the type Ty is a class that has a trivial copy constructor, otherwise it holds false.

A **copy constructor** for a class Ty is **trivial** if:
- it is implicitly declared
- the class Ty has no virtual functions
- the class Ty has no virtual bases
- all the direct bases of the class Ty have trivial copy constructors
- the classes of all the non-static data members of class type have trivial copy constructors
- the classes of all the non-static data members of type array of class have trivial copy constructors

has_trivial_destructor:

```
template <class Ty>
    struct has_trivial_destructor;
```

An instance of the type predicate holds true if the type Ty is a class that has a trivial destructor, otherwise it holds false.

A **destructor** for a class Ty is **trivial** if:
- it is an implicitly declared destructor
- all the direct bases of the class Ty have trivial destructors
- the classes of all the non-static data members of class type have trivial destructors
- the classes of all the non-static data members of type array of class have trivial destructors

has_nothrow_constructor:

```
template <class Ty>
    struct has_nothrow_constructor;
```

An instance of the type predicate holds true if the type Ty has a nothrow default constructor, otherwise it holds false.

has_nothrow_copy:

```
template <class Ty>
    struct has_nothrow_copy;
```

An instance of the type predicate holds true if the type Ty has a nothrow copy constructor, otherwise it holds false.

has_nothrow_assign:

```
template <class Ty>
    struct has_nothrow_assign;
```

An instance of the type predicate holds true if the type Ty has a nothrow copy assignment operator, otherwise it holds false.

A **nothrow** function is a function that has an empty throw specifier, or a function which the compiler can otherwise determine will not throw an exception.

has_virtual_destructor:

```
template <class Ty>
    struct has_virtual_destructor;
```

An instance of the type predicate holds true if the type Ty is a class that has a virtual destructor, otherwise it holds false.

is_signed:

```
template <class _Ty>
    struct is_signed;
```

An instance of the type predicate holds true if the type Ty is a signed integral type or a cv-qualified signed integral type, otherwise it holds false.

is_unsigned:

```
template <class Ty>
    struct is_unsigned;
```

An instance of the type predicate holds true if the type Ty is an unsigned integral type or a cv-qualified unsigned integral type, otherwise it holds false.

alignment_of:

```
template <class Ty>
    struct alignment_of;
```

The type query holds the value of the alignment of the type Ty.

rank:

```
template <class Ty>
    struct rank;
```

The type query holds the value of the number of dimensions of the array type Ty, or 0 if Ty is not an array type.

extent:

```
template <class _Ty, unsigned _I = 0> struct extent;
```

The type query holds the value of the number of elements in the I^{th} bound of objects of type `Ty`. If `Ty` is not an array type or its rank is less than `I`, or if `I` is zero and `Ty` is of type "array of unknown bound of U", it holds the value 0.

Binary Type Traits

Binary Type Traits provide information about a relationship between two types. Every Binary Type Trait possesses a static data member of type `bool` named `value`. This member indicates the presence or absence of a specific relationship between the two argument types. For example, the value of the following expression will be true if the type arguments `Ty1` and `Ty2` are the same type, and false otherwise:

```
std::tr1::is_same<Ty1, Ty2>::value
```

is_same

```
template <class Ty, class Ty2>
    struct is_same;
```

An instance of the type predicate holds true if the types `Ty1` and `Ty2` are the same type, otherwise it holds false.

is_convertible

```
template <class From, class To>
    struct is_convertible;
```

An instance of the type predicate holds true if the expression `To to = from;`, where `from` is an object of type `From`, is well-formed.

is_base_of

```
template <class Base, class Derived>
    struct is_base_of;
```

An instance of the type predicate holds true if the type `Base` is a base class of the type `Derived`, otherwise it holds false.

Transformation Type Traits

Transformation Type Traits modify a type. Every Transformation Type Trait possesses a nested typedef named `type` that represents the result of the modification. For example, for a given type `Ty`, the type `std::tr1::add_reference<Ty>::type` is equivalent to the type `Ty &`.

remove_const

```
template <class Ty>
    struct remove_const;
```

An instance of the type modifier holds a modified-type that is `Ty1` when `Ty` is of the form `const Ty1`, otherwise `Ty`.

remove_volatile

```
template <class Ty>
    struct remove_volatile;
```

An instance of the type modifier holds a modified-type that is `Ty1` when `Ty` is of the form `volatile Ty1`, otherwise `Ty`.

remove_cv

```
template <class Ty>
    struct remove_cv;
```

An instance of the type modifier holds a modified-type that is Ty1 when Ty is of the form const Ty1, volatile Ty1, or const volatile Ty1, otherwise Ty.

add_const

```
template <class Ty>
    struct add_const;
```

An instance of the type modifier holds a modified-type that is Ty if Ty is a reference, a function, or a const-qualified type, otherwise const Ty.

add_volatile

```
template <class Ty>
    struct add_volatile;
```

An instance of the type modifier holds a modified-type that is Ty if Ty is a reference, a function, or a volatile-qualified type, otherwise volatile volatile Ty.

add_cv

```
template <class Ty>
    struct add_cv;
```

An instance of the type modifier holds the modified-type add_volatile< add_const<Ty> >.

remove_reference

```
template <class Ty>
    struct remove_reference;
```

An instance of the type modifier holds a modified-type that is Ty1 when Ty is of the form Ty1&, otherwise Ty.

add_reference

```
template <class Ty>
    struct add_reference;
```

An instance of the type modifier holds a modified-type that is Ty if Ty is a reference, otherwise Ty&.

remove_pointer

```
template <class Ty>
    struct remove_pointer;
```

An instance of the type modifier holds a modified-type that is Ty1 when Ty is of the form Ty1*, Ty1* const, Ty1* volatile, or Ty1* const volatile, otherwise Ty.

add_pointer

```
template <class Ty>
    struct add_pointer;
```

An instance of the type modifier holds the modified-type Ty1* if Ty is of the form Ty1[N]or Ty1&, otherwise Ty*.

remove_extent

```
template <class Ty>
    struct remove_extent;
```

An instance of the type modifier holds a modified-type that is Ty1 when Ty is of the formTy1[N], otherwise Ty.

remove_all_extents

```
template <class Ty>
    struct remove_all_extents;
```

An instance of the type modifier holds a modified-type that is the element type of the array type Ty with all array dimensions removed, or Ty if Ty is not an array type.

aligned_storage

```
template <std::size_t _Len, std::size_t _Align>
    struct aligned_storage {
        typedef aligned-type type;
        };
```

The nested typedef type is a synonym for a POD type with alignment Align and size Len. Align must be equal to alignment_of<Ty1>::valuefor some type Ty1.

<unordered_map>

Description

Include the STL standard header **<unordered_map>** to define the container template classes unordered_map and unordered_multimap, and their supporting templates.

Note: To enable this header file, you must define the macro __IBMCPP_TR1__ and use the **TARGET** compiler option to specify a valid release level. Valid release levels for TR1 support are zOSV1R12 or later. For example, you can specify the **TARGET** option as follows: TARGET(zOSV1R12).

Any release prior to zOSV1R12 is invalid for the use of TR1. If TR1 code is used in an application to be compiled on an earlier platform, the compiler will issue the #error directive.

Synopsis

```
namespace std {
    namespace tr1 {
    // DECLARATIONS
template <class Key, class Ty, class Hash, class Pred, class Alloc>
    class unordered_map;
template <class Key, class Ty, class Hash, class Pred, class Alloc>
    class unordered_multimap;

    // TEMPLATE FUNCTIONS
template <class Key, class Ty, class Hash, class Pred, class Alloc>
    void swap(
        unordered_map<Key, Ty, Hash, Pred, Alloc>& left,
        unordered_map<Key, Ty, Hash, Pred, Alloc>& right);
template <class Key, class Ty, class Hash, class Pred, class Alloc>
    void swap(
```

```
        unordered_multimap<Key, Ty, Hash, Pred, Alloc>& left,
        unordered_multimap<Key, Ty, Hash, Pred, Alloc>& right);
    } // namespace tr1
} // namespace std
```

Classes

unordered_map

Description: The template class describes an object that controls a varying-length sequence of elements of type `std::pair<const Key, Ty>`. The sequence is weakly ordered by a **hash function**, which partitions the sequence into an ordered set of subsequences called **buckets**. Within each bucket a **comparison function** determines whether any pair of elements has **equivalent ordering**. Each element stores two objects, a **sort key** and a **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling two stored objects, a comparison function object of type `key_equal` and a hash function object of type `hasher`. You access the first stored object by calling the member function `key_eq()`; and you access the second stored object by calling the member function `hash_function()`. Specifically, for all values `X` and `Y` of type `Key`, the call `key_eq()(X, Y)` returns true only if the two argument values have equivalent ordering; the call `hash_function()(keyval)` yields a distribution of values of type `size_t`. Unlike template class `unordered_multimap`, an object of template class `unordered_map` ensures that `key_eq()(X, Y)` is always false for any two elements of the controlled sequence. (Keys are unique.)

The object also stores a **maximum load factor**, which specifies the maximum desired average number of elements per bucket. If inserting an element causes `load_factor()` to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You cannot in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored **allocator object** of type `allocator_type`. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that the stored allocator object is *not* copied when the container object is assigned.

Synopsis:

```
template <class Key,
    class Ty,
    class Hash  = std::tr1::hash<Key>,
    class Pred  = std::equal_to<Key>,
    class Alloc= std::allocator<Key> >
```

```
    class unordered_map {
public:
    typedef Key key_type;
    typedef Ty mapped_type;
    typedef std::pair<const Key, Ty> value_type;
    typedef Hash hasher;
    typedef Pred key_equal;
    typedef Alloc allocator_type;

    typedef Alloc::pointer pointer;
    typedef Alloc::const_pointer const_pointer;
    typedef Alloc::reference reference;
    typedef Alloc::const_reference const_reference;

    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef T4 local_iterator;
    typedef T5 const_local_iterator;

    unordered_map(
        const unordered_map& right);
    explicit unordered_map(
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());
    template <class InIt>
        unordered_map(
        InIt first, InIt last,
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());

    iterator begin();
    const_iterator begin() const;
    local_iterator begin(size_type nbucket);
    const_local_iterator begin(size_type nbucket) const;

    iterator end();
    const_iterator end() const;
    local_iterator end(size_type nbucket);
    const_local_iterator end(size_type nbucket) const;

    size_type bucket_count() const;
    size_type max_bucket_count() const;
    size_type bucket(const Key& keyval) const;
    size_type bucket_size(size_type nbucket) const;

    Hash hash_function() const;
    Pred key_eq() const;
    Alloc get_allocator() const;

    float load_factor() const;
    float max_load_factor() const;
    void max_load_factor(float factor);
    void rehash(size_type nbuckets);

    Ty operator[](const Key& keyval);

    std::pair<iterator, bool> insert(const value_type& val);
    iterator insert(iterator where, const value_type& val);
    template<class InIt>
        void insert(InIt first, InIt last);
```

```
    iterator erase(iterator where);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& keyval);
    void clear();

    void swap(unordered_map& right);

    const_iterator find(const Key& keyval) const;
    size_type count(const Key& keyval) const;
    std::pair<iterator, iterator>
        equal_range(const Key& keyval);
    std::pair<const_iterator, const_iterator>
        equal_range(const Key& keyval) const;
    };
```

Constructor:

unordered_map::unordered_map:

```
unordered_map(
    const unordered_map& right);
explicit unordered_map(
   size_type nbuckets = N0,
    const Hash& hfn = Hash(),
    const Pred& comp = Pred(),
    const Alloc& al = Alloc());
template <class InIt>
  unordered_map(
    InIt first, InIt last,
    size_type nbuckets = N0,
    const Hash& hfn = Hash(),
    const Pred& comp = Pred(),
    const Alloc& al = Alloc());
```

The first constructor specifies a copy of the sequence controlled by `right`. The second constructor specifies an empty controlled sequence. The third constructor inserts the sequence of element values `[first, last)`.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from `right`. Otherwise:

- the minimum number of buckets is the argument `nbuckets`, if present; otherwise it is a default value described here as the implementation-defined value `N0`.
- the hash function object is the argument `hfn`, if present; otherwise it is `Hash()`.
- the comparison function object is the argument `comp`, if present; otherwise it is `Pred()`.
- the allocator object is the argument `al`, if present; otherwise, it is `Alloc()`.

Types:

unordered_map::allocator_type:

```
typedef Alloc allocator_type;
```

The type is a synonym for the template parameter `Alloc`.

unordered_map::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

unordered_map::const_local_iterator:

```
typedef T5 const_local_iterator;
```

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T5`.

unordered_map::const_pointer:

```
typedef Alloc::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

unordered_map::const_reference:

```
typedef Alloc::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

unordered_map::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

unordered_map::hasher:

```
typedef Hash hasher;
```

The type is a synonym for the template parameter `Hash`.

unordered_map::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

unordered_map::key_equal:

```
typedef Pred key_equal;
```

The type is a synonym for the template parameter `Pred`.

unordered_map::key_type:

```
typedef Key key_type;
```

The type is a synonym for the template parameter `Key`.

unordered_map::local_iterator:

```
typedef T4 local_iterator;
```

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T4`.

unordered_map::mapped_type:

```
typedef Ty mapped_type;
```

The type is a synonym for the template parameter Ty.

unordered_map::pointer:

```
typedef Alloc::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

unordered_map::reference:

```
typedef Alloc::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

unordered_map::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

unordered_map::value_type:

```
typedef std::pair<const Key, Ty> value_type;
```

The type describes an element of the controlled sequence.

Member functions:

unordered_map::begin:

```
iterator begin();
    const_iterator begin() const;
    local_iterator begin(size_type nbucket);
    const_local_iterator begin(size_type nbucket) const;
```

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket nbucket (or just beyond the end of an empty bucket).

unordered_map::bucket:

```
    size_type bucket(const Key& keyval) const;
```

The member function returns the bucket number currently corresponding to the key value keyval.

unordered_map::bucket_count:

```
    size_type bucket_count() const;
```

The member function returns the current number of buckets.

unordered_map::bucket_size:

```
    size_type bucket_size(size_type nbucket) const;
```

The member function returns the size of bucket number nbucket.

unordered_map::clear:

```
void clear();
```

The member function calls `erase( begin(), end())`.

unordered_map::count:

```
size_type count(const Key& keyval) const;
```

The member function returns the number of elements in the range delimited by `equal_range(keyval)`.

unordered_map::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

unordered_map::end:

```
iterator end();
    const_iterator end() const;
    local_iterator end(size_type nbucket);
    const_local_iterator end(size_type nbucket) const;
```

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket `nbucket`.

unordered_map::equal_range:

```
std::pair<iterator, iterator>
    equal_range(const Key& keyval);
std::pair<const_iterator, const_iterator>
    equal_range(const Key& keyval) const;
```

The member function returns a pair of iterators `X` such that `[X.first, X.second)` delimits just those elements of the controlled sequence that have equivalent ordering with `keyval`. If no such elements exist, both iterators are `end()`.

unordered_map::erase:

```
iterator erase(iterator where);
iterator erase(iterator first, iterator last);
size_type erase(const Key& keyval);
```

The first member function removes the element of the controlled sequence pointed to by `where`. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes the elements in the range delimited by `equal_range(keyval)`. It returns the number of elements it removes.

The member functions never throw an exception.

unordered_map::find:

```
const_iterator find(const Key& keyval) const;
```

The member function returns `equal_range(keyval).first`.

unordered_map::get_allocator:

```
Alloc get_allocator() const;
```

The member function returns the stored allocator object.

unordered_map::hash_function:

```
Hash hash_function() const;
```

The member function returns the stored hash function object.

unordered_map::insert:

```
std::pair<iterator, bool> insert(const value_type& val);
iterator insert(iterator where, const value_type& val);
template<class InIt>
    void insert(InIt first, InIt last);
```

The first member function determines whether an element `X` exists in the sequence whose key has equivalent ordering to that of `val`. If not, it creates such an element `X` and initializes it with `val`. The function then determines the iterator `where` that designates `X`. If an insertion occurred, the function returns `std::pair(where, true)`. Otherwise, it returns `std::pair(where, false)`.

The second member function returns `insert(val).first`, using `where` as a starting place within the controlled sequence to search for the insertion point. (Insertion can possibly occur somewhat faster, if the insertion point immediately precedes or follows `where`.) The third member function inserts the sequence of element values, for each `where` in the range `[first, last)`, by calling `insert(*where)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

unordered_map::key_eq:

```
Pred key_eq() const;
```

The member function returns the stored comparison function object.

unordered_map::load_factor:

```
float load_factor() const;
```

The member function returns `(float)size() / (float)bucket_count()`, the average number of elements per bucket.

unordered_map::max_bucket_count:

```
size_type max_bucket_count() const;
```

The member function returns the maximum number of buckets currently permitted.

unordered_map::max_load_factor:

```
float max_load_factor() const;
void max_load_factor(float factor);
```

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with factor.

unordered_map::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

unordered_map::operator[]:

```
Ty& operator[](const Key& keyval);
```

The member function determines the iterator `where` as the return value of `insert( value_type(keyval, Ty())`. (It inserts an element with the specified key if no such element exists.) It then returns a reference to `(*where).second`.

unordered_map::rehash:

```
void rehash(size_type nbuckets);
```

The member function alters the number of buckets to be at least `nbuckets` and rebuilds the hash table as needed.

unordered_map::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

unordered_map::swap:

```
void swap(unordered_map& right);
```

The member function swaps the controlled sequences between `*this` and `right`. If `get_allocator() == right.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored traits object of type`Tr`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

unordered_multimap

Description: The template class describes an object that controls a varying-length sequence of elements of type `std::pair<const Key, Ty>`. The sequence is weakly ordered by a **hash function**, which partitions the sequence into an ordered set of subsequences called **buckets**. Within each bucket a **comparison function** determines whether any pair of elements has **equivalent ordering**. Each element stores two objects, a **sort key** and a **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling two stored objects, a comparison function object of type `key_equal` and a hash function object of type `hasher`. You access the first stored object by calling the member function `key_eq()`; and you access the second stored object by calling the member function `hash_function()`. Specifically, for all values `X` and `Y` of type `Key`, the call `key_eq()(X, Y)` returns true only if the two argument values have equivalent ordering; the call `hash_function()(keyval)` yields a distribution of values of type `size_t`. Unlike template class `unordered_map`, an object of template class unordered_multimap does not ensure that key_eq()(X, Y) is always false for any two elements of the controlled sequence. (Keys need not be unique.)

The object also stores a **maximum load factor**, which specifies the maximum desired average number of elements per bucket. If inserting an element causes load_factor() to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You cannot in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored **allocator object** of type allocator_type. Such an allocator object must have the same external interface as an object of template class allocator. Note that the stored allocator object is *not* copied when the container object is assigned.

Synopsis:

```
template <class Key,
    class Ty,
    class Hash  = std::tr1::hash<Key>,
    class Pred  = std::equal_to<Key>,
    class Alloc= std::allocator<Key> >
    class unordered_multimap {
public:
    typedef Key key_type;
    typedef Ty mapped_type;
    typedef std::pair<const Key, Ty> value_type;
    typedef Hash hasher;
    typedef Pred key_equal;
    typedef Alloc allocator_type;

    typedef Alloc::pointer pointer;
    typedef Alloc::const_pointer const_pointer;
    typedef Alloc::reference reference;
    typedef Alloc::const_reference const_reference;

    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef T4 local_iterator;
    typedef T5 const_local_iterator;

    unordered_multimap(
        const unordered_multimap& right);
    explicit unordered_multimap(
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());
    template <class InIt>
        unordered_multimap(
        InIt first, InIt last,
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());

    iterator begin();
```

```
    const_iterator begin() const;
    local_iterator begin(size_type nbucket);
    const_local_iterator begin(size_type nbucket) const;

    iterator end();
    const_iterator end() const;
    local_iterator end(size_type nbucket);
    const_local_iterator end(size_type nbucket) const;

    size_type size() const;
    size_type max_size() const;
    bool empty() const;

    size_type bucket_count() const;
    size_type max_bucket_count() const;
    size_type bucket(const Key& keyval) const;
    size_type bucket_size(size_type nbucket) const;

    Hash hash_function() const;
    Pred key_eq() const;
    Alloc get_allocator() const;

    float load_factor() const;
    float max_load_factor() const;
    void max_load_factor(float factor);
    void rehash(size_type nbuckets);

    iterator insert(const value_type& val);
    iterator insert(iterator where, const value_type& val);
    template<class InIt>
        void insert(InIt first, InIt last);

    iterator erase(iterator where);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& keyval);
    void clear();

    void swap(unordered_multimap& right);

    const_iterator find(const Key& keyval) const;
    size_type count(const Key& keyval) const;
    std::pair<iterator, iterator>
        equal_range(const Key& keyval);
    std::pair<const_iterator, const_iterator>
        equal_range(const Key& keyval) const;
    };
```

Constructor:

unordered_multimap::unordered_multimap:

```
unordered_multimap(
    const unordered_multimap& right);
explicit unordered_multimap(
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());
template <class InIt>
        unordered_multimap(
        InIt first, InIt last,
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());
```

The first constructor specifies a copy of the sequence controlled by `right`. The second constructor specifies an empty controlled sequence. The third constructor inserts the sequence of element values `[first, last)`.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from `right`. Otherwise:

- the minimum number of buckets is the argument `nbuckets`, if present; otherwise it is a default value described here as the implementation-defined value `N0`.
- the hash function object is the argument `hfn`, if present; otherwise it is `Hash()`.
- the comparison function object is the argument `comp`, if present; otherwise it is `Pred()`.
- the allocator object is the argument `al`, if present; otherwise, it is `Alloc()`.

Types:

unordered_multimap::allocator_type:

```
typedef Alloc allocator_type;
```

The type is a synonym for the template parameter `Alloc`.

unordered_multimap::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

unordered_multimap::const_local_iterator:

```
typedef T5 const_local_iterator;
```

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T5`.

unordered_multimap::const_pointer:

```
typedef Alloc::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

unordered_multimap::const_reference:

```
typedef Alloc ::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

unordered_multimap::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

unordered_multimap::hasher:

```
typedef Hash hasher;
```

The type is a synonym for the template parameter Hash.

unordered_multimap::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T0.

unordered_multimap::key_equal:

```
typedef Pred key_equal;
```

The type is a synonym for the template parameter Pred.

unordered_multimap::key_type:

```
typedef Key key_type;
```

The type is a synonym for the template parameter Key.

unordered_multimap::local_iterator:

```
typedef T4 local_iterator;
```

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type T4.

unordered_multimap::mapped_type:

```
typedef Ty mapped_type;
```

The type is a synonym for the template parameter Ty.

unordered_multimap::pointer:

```
typedef Alloc::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

unordered_multimap::reference:

```
typedef Alloc::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

unordered_multimap::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

unordered_multimap::value_type:

```
typedef std::pair<const Key, Ty> value_type;
```

The type describes an element of the controlled sequence.

Member functions:

unordered_multimap::begin:

```
iterator begin();
    const_iterator begin() const;
    local_iterator begin(size_type nbucket);
    const_local_iterator begin(size_type nbucket) const;
```

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket `nbucket` (or just beyond the end of an empty bucket).

unordered_multimap::bucket:

```
    size_type bucket(const Key& keyval) const;
```

The member function returns the bucket number currently corresponding to the key value `keyval`.

unordered_multimap::bucket_count:

```
    size_type bucket_count() const;
```

The member function returns the current number of buckets.

unordered_multimap::bucket_size:

```
    size_type bucket_size(size_type nbucket) const;
```

The member functions returns the size of bucket number `nbucket`.

unordered_multimap::clear:

```
void clear();
```

The member function calls `erase( begin(), end())`.

unordered_multimap::count:

```
size_type count(const Key& keyval) const;
```

The member function returns the number of elements in the range delimited by `equal_range(keyval)`.

unordered_multimap::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

unordered_multimap::end:

```
iterator end();
    const_iterator end() const;
    local_iterator end(size_type nbucket);
    const_local_iterator end(size_type nbucket) const;
```

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket `nbucket`.

unordered_multimap::equal_range:

```
std::pair<iterator, iterator>
  equal_range(const Key& keyval);
std::pair<const_iterator, const_iterator>
  equal_range(const Key& keyval) const;
```

The member function returns a pair of iterators `X` such that `[X.first, X.second)` delimits just those elements of the controlled sequence that have equivalent ordering with `keyval`. If no such elements exist, both iterators are `end()`.

unordered_multimap::erase:

```
iterator erase(iterator where);
iterator erase(iterator first, iterator last);
sizetype erase(const Key& keyval);
```

The first member function removes the element of the controlled sequence pointed to by `where`. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes the elements in the range delimited by `equal_range(keyval)`. It returns the number of elements it removes.

The member functions never throw an exception.

unordered_multimap::find:

```
const_iterator find(const Key& keyval) const;
```

The member function returns `equal_range(keyval).first`.

unordered_multimap::get_allocator:

```
Alloc get_allocator() const;
```

The member function returns the stored allocator object.

unordered_multimap::hash_function:

```
Hash hash_function() const;
```

The member function returns the stored hash function object.

unordered_multimap::insert:

```
iterator insert(const value_type& val);
iterator insert(iterator where, const value_type& val);
template <class InIt>
   void insert(InIt first, InIt last);
```

The first member function inserts the element `val` in the controlled sequence, then returns the iterator that designates the inserted element. The second member function returns `insert(val)`, using `where` as a starting place within the controlled sequence to search for the insertion point. (Insertion can possibly occur somewhat faster, if the insertion point immediately precedes or follows `where`.) The third member function inserts the sequence of element values, for each `where` in the range **[first, last)**, by calling `insert(*where)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

unordered_multimap::key_eq:

```
Pred key_eq() const;
```

The member function returns the stored comparison function object.

unordered_multimap::load_factor:

```
float load_factor() const;
```

The member function returns `(float)size() / (float)bucket_count()`, the average number of elements per bucket.

unordered_multimap::max_bucket_count:

```
size_type max_bucket_count() const;
```

The member function returns the maximum number of buckets currently permitted.

unordered_multimap::max_load_factor:

```
float max_load_factor() const;
void max_load_factor(float z);
```

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with `factor`.

unordered_multimap::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

unordered_multimap::rehash:

```
void rehash(size_type nbuckets);
```

The member function alters the number of buckets to be at least `nbuckets` and rebuilds the hash table as needed.

unordered_multimap::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

unordered_multimap::swap:

```
void swap(unordered_multimap& right);
```

The member function swaps the controlled sequences between `*this` and `right`. If `get_allocator() == right.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type `Tr`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

<unordered_set>

Description

Include the STL standard header **<unordered_set>** to define the container template classes `unordered_set` and `unordered_multiset`, and their supporting templates.

Note: To enable this header file, you must define the macro __IBMCPP_TR1__ and use the **TARGET** compiler option to specify a valid release level. Valid release levels for TR1 support are zOSV1R12 or later. For example, you can specify the **TARGET** option as follows: `TARGET(zOSV1R12)`.

Any release prior to zOSV1R12 is invalid for the use of TR1. If TR1 code is used in an application to be compiled on an earlier platform, the compiler will issue the `#error` directive.

Synopsis

```
namespace std {
namespace tr1 {
      // DECLARATIONS
  template <class Key, class Hash, class Pred, class Alloc>
  class unordered_set;
  template <class Key, class Hash, class Pred, class Alloc>
  class unordered_multiset;

     // TEMPLATE FUNCTIONS
  template <class Key, class Hash, class Pred, class Alloc>
  void swap(
      unordered_set<Key, Hash, Pred, Alloc>& left,
      unordered_set<Key, Hash, Pred, Alloc>& right);
  template <class Key, class Hash, class Pred, class Alloc>
  void swap(
      unordered_multiset<Key, Hash, Pred, Alloc>& left,
      unordered_multiset<Key, Hash, Pred, Alloc>& right);
} // namespace tr1
} // namespace std
```

Classes

unordered_multiset

Description: The template class describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is weakly ordered by a **hash function**, which partitions the sequence into an ordered set of subsequences called **buckets**. Within each bucket a **comparison function** determines whether any pair of elements has **equivalent ordering**. Each element serves as both a **sort key** and a **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling two stored objects, a comparison function object of type `key_equal` and a hash function object of type `hasher`. You access the first stored object by calling the member function `key_eq()`;

and you access the second stored object by calling the member function hash_function(). Specifically, for all values X and Y of type Key, the call key_eq()(X, Y) returns true only if the two argument values have equivalent ordering; the call hash_function()(keyval) yields a distribution of values of type size_t. Unlike template class unordered_set, an object of template class unordered_multiset does not ensure that key_eq()(X, Y) is always false for any two elements of the controlled sequence.

The object also stores a **maximum load factor**, which specifies the maximum desired average number of elements per bucket. If inserting an element causes load_factor() to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You cannot in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored allocator object of type allocator_type. Such an allocator object must have the same external interface as an object of template class allocator. Note that the stored allocator object is *not* copied when the container object is assigned.

Synopsis:

```
template <class Key,
    class Hash  = std::tr1::hash<Key>,
    class Pred  = std::equal_to<Key>,
    class Alloc= std::allocator<Key> >
    class unordered_multiset {
public:
    typedef Key key_type;
    typedef Key value_type;
    typedef Hash hasher;
    typedef Pred key_equal;
    typedef Alloc allocator_type;

    typedef Alloc::pointer pointer;
    typedef Alloc::const_pointer const_pointer;
    typedef Alloc::reference reference;
    typedef Alloc::const_reference const_reference;

    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef T4 local_iterator;
    typedef T5 const_local_iterator;

    unordered_multiset {
        const unordered_multiset& right);
    explicit unordered_multiset(
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());
    template<class InIt>
        unordered_multiset(
        InIt first, InIt last,
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
```

```
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());

    iterator begin();
    const_iterator begin() const;
    local_iterator begin(size_type nbucket);
    const_local_iterator begin(size_type nbucket) const;

    iterator end();
    const_iterator end() const;
    local_iterator end(size_type nbucket);
    const_local_iterator end(size_type nbucket) const;

    size_type size() const;
    size_type max_size() const;
    bool empty() const;

    size_type bucket_count() const;
    size_type max_bucket_count() const;
    size_type bucket(const Key& keyval) const;
    size_type bucket_size(size_type nbucket) const;

    Hash hash_function() const;
    Pred key_eq() const;
    Alloc get_allocator() const;

    float load_factor() const;
    float max_load_factor() const;
    void max_load_factor(float factor);
    void rehash(size_type nbuckets);

    std::pair<iterator, bool> insert(const value_type& val);
    iterator insert(iterator where, const value_type& val);
    template<class InIt>
        void insert(InIt first, InIt last);

    iterator erase(iterator where);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& keyval);
    void clear();

    void swap(unordered_multiset& right);

    const_iterator find(const Key& keyval) const;
    size_type count(const Key& keyval) const;
    std::pair<iterator, iterator>
        equal_range(const Key& keyval);
    std::pair<const_iterator, const_iterator>
        equal_range(const Key& keyval) const;
    };
```

Constructor:

unordered_multiset::unordered_multiset:

```
    unordered_multiset {
        const unordered_multiset& right);
    explicit unordered_multiset(
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());
    template<class InIt>
        unordered_multiset(
        InIt first, InIt last,
```

```
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());
```

The first constructor specifies a copy of the sequence controlled by `right`. The second constructor specifies an empty controlled sequence. The third constructor inserts the sequence of element values `[first, last)`.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from `right`. Otherwise:
- the minimum number of buckets is the argument `nbuckets`, if present; otherwise it is a default value described here as the implementation-defined value `N0`.
- the hash function object is the argument `hfn`, if present; otherwise it is `Hash()`.
- the comparison function object is the argument `comp`, if present; otherwise it is `Pred()`.
- the allocator object is the argument `al`, if present; otherwise, it is `Alloc()`.

Types:

unordered_multiset::allocator_type:

```
typedef Alloc allocator_type;
```

The type is a synonym for the template parameter `Alloc`.

unordered_multiset::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

unordered_multiset::const_local_iterator:

```
typedef T5 const_local_iterator;
```

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T5`.

unordered_multiset::const_pointer:

```
typedef Alloc::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

unordered_multiset::const_reference:

```
typedef Alloc::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

unordered_multiset::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

unordered_multiset::hasher:

```
typedef Hash hasher;
```

The type is a synonym for the template parameter Hash.

unordered_multiset::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T0.

unordered_multiset::key_equal:

```
typedef Pred key_equal;
```

The type is a synonym for the template parameter Pred.

unordered_multiset::key_type:

```
typedef Key key_type;
```

The type is a synonym for the template parameter Key.

unordered_multiset::local_iterator:

```
typedef T4 local_iterator;
```

The type describes an object that can serve as a forward iterator for a bucket. It is described here as a synonym for the implementation-defined type T4.

unordered_multiset::pointer:

```
typedef Alloc::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

unordered_multiset::reference:

```
typedef Alloc::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

unordered_multiset::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

unordered_multiset::value_type:

```
typedef Key value_type;
```

The type describes an element of the controlled sequence.

Member functions:

unordered_multiset::begin:

```
iterator begin();
    const_iterator begin() const;
    local_iterator begin(size_type nbucket);
    const_local_iterator begin(size_type nbucket) const;
```

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket `nbucket` (or just beyond the end of an empty bucket).

unordered_multiset::bucket:

```
    size_type bucket(const Key& keyval) const;
```

The member function returns the bucket number currently corresponding to the key value `keyval`.

unordered_multiset::bucket_count:

```
    size_type bucket_count() const;
```

The member function returns the current number of buckets.

unordered_multiset::bucket_size:

```
    size_type bucket_size(size_type nbucket) const;
```

The member functions returns the size of bucket number `nbucket`.

unordered_multiset::clear:

```
void clear();
```

The member function calls `erase( begin(), end())`.

unordered_multiset::count:

```
size_type count(const Key& keyval) const;
```

The member function returns the number of elements in the range delimited by `equal_range(keyval)`.

unordered_multiset::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

unordered_multiset::end:

```
iterator end();
    const_iterator end() const;
    local_iterator end(size_type nbucket);
    const_local_iterator end(size_type nbucket) const;
```

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket `nbucket`.

unordered_multiset::equal_range:

```
std::pair<iterator, iterator>
  equal_range(const Key& keyval);
std::pair<const_iterator, const_iterator>
  equal_range(const Key& keyval) const;
```

The member function returns a pair of iterators `X` such that `[X.first, X.second)` delimits just those elements of the controlled sequence that have equivalent ordering with `keyval`. If no such elements exist, both iterators are `end()`.

unordered_multiset::erase:

```
iterator erase(iterator where);
iterator erase(iterator first, iterator last);
size_type erase(const Key& keyval);
```

The first member function removes the element of the controlled sequence pointed to by `where`. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes the elements in the rang delimited by `equal_range(keyval)`. It returns the number of elements it removes.

The member functions never throw an exception.

unordered_multiset::find:

```
const_iterator find(const Key& keyval) const;
```

The member function returns `equal_range(keyval).first`.

unordered_multiset::get_allocator:

```
Alloc get_allocator() const;
```

The member function returns the stored allocator object.

unordered_multiset::hash_function:

```
  Hash hash_function() const;
```

The member function returns the stored hash function object.

unordered_multiset::insert:

```
    std::pair<iterator, bool> insert(const value_type& val);
    iterator insert(iterator where, const value_type& val);
    template<class InIt>
        void insert(InIt first, InIt last);
```

The first member function determines whether an element `X` exists in the sequence whose key has **equivalent ordering** to that of `val`. If not, it creates such an element `X` and initializes it with `val`. The function then determines the iterator where that designates `X`. If an insertion occurred, the function returns `std::pair(where, true)`. Otherwise, it returns `std::pair(where, false)`.

The second member function returns `insert(val).first`, using `where` as a starting place within the controlled sequence to search for the insertion point. (Insertion can possibly occur somewhat faster, if the insertion point immediately precedes or follows `where`.) The third member function inserts the sequence of element values, for each `where` in the range `[first, last)`, by calling `insert(*where)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

unordered_multiset::key_eq:

```
Pred key_eq() const;
```

The member function returns the stored comparison function object.

unordered_multiset::load_factor:

```
float load_factor() const;
```

The member function returns `(float)size() / (float)bucket_count()`, the average number of elements per bucket.

unordered_multiset::max_bucket_count:

```
size_type max_bucket_count() const;
```

The member function returns the maximum number of buckets currently permitted.

unordered_multiset::max_load_factor:

```
float max_load_factor() const;
void max_load_factor(float factor);
```

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with `factor`.

unordered_multiset::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

unordered_multiset::rehash:

```
void rehash(size_type nbuckets);
```

The member function alters the number of buckets to be at least `nbuckets` and rebuilds the hash table as needed.

unordered_multiset::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

unordered_multiset::swap:

```
void swap(unordered_multiset& right);
```

The member function swaps the controlled sequences between `*this` and `right`. If `get_allocator() == right.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type `Tr`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

unordered_set

Description: The template class describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is weakly ordered by a **hash function**, which partitions the sequence into an ordered set of subsequences called **buckets**. Within each bucket a **comparison function** determines whether any pair of elements has **equivalent ordering**. Each element serves as both a **sort key** and a **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling two stored objects, a comparison function object of type `key_equal` and a hash function object of type `hasher`. You access the first stored object by calling the member function `key_eq()`; and you access the second stored object by calling the member function `hash_function()`. Specifically, for all values `X` and `Y` of type `Key`, the call `key_eq()(X, Y)` returns true only if the two argument values have equivalent ordering; the call `hash_function()(keyval)` yields a distribution of values of type `size_t`. Unlike template class `unordered_multiset`, an object of template class `unordered_set` ensures that `key_eq()(X, Y)` is always false for any two elements of the controlled sequence.

The object also stores a **maximum load factor**, which specifies the maximum desired average number of elements per bucket. If inserting an element causes `load_factor()` to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You cannot in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored allocator object of type `allocator_type`. Such an allocator object must have the same external interface as an object of template class allocator. Note that the stored allocator object is *not* copied when the container object is assigned.

Synopsis:

```
template <class Key,
    class Hash  = std::tr1::hash<Key>,
    class Pred  = std::equal_to<Key>,
    class Alloc= std::allocator<Key> >
    class unordered_set {
public:
    typedef Key key_type;
    typedef Key value_type;
    typedef Hash hasher;
```

```
typedef Pred key_equal;
typedef Alloc allocator_type;

typedef Alloc::pointer pointer;
typedef Alloc::const_pointer const_pointer;
typedef Alloc::reference reference;
typedef Alloc::const_reference const_reference;

typedef T0 iterator;
typedef T1 const_iterator;
typedef T2 size_type;
typedef T3 difference_type;
typedef T4 local_iterator;
typedef T5 const_local_iterator;

unordered_set {
    const unordered_set& right);
explicit unordered_set(
    size_type nbuckets = N0,
    const Hash& hfn = Hash(),
    const Pred& comp = Pred(),
    const Alloc& al = Alloc());
template<class InIt>
    unordered_set(
    InIt first, InIt last,
    size_type nbuckets = N0,
    const Hash& hfn = Hash(),
    const Pred& comp = Pred(),
    const Alloc& al = Alloc());

iterator begin();
const_iterator begin() const;
local_iterator begin(size_type nbucket);
const_local_iterator begin(size_type nbucket) const;

iterator end();
const_iterator end() const;
local_iterator end(size_type nbucket);
const_local_iterator end(size_type nbucket) const;

size_type size() const;
size_type max_size() const;
bool empty() const;

size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket(const Key& keyval) const;
size_type bucket_size(size_type nbucket) const;

Hash hash_function() const;
Pred key_eq() const;
Alloc get_allocator() const;

float load_factor() const;
float max_load_factor() const;
void max_load_factor(float factor);
void rehash(size_type nbuckets);

iterator insert(const value_type& val);
iterator insert(iterator where, const value_type& val);
template<class InIt>
    void insert(InIt first, InIt last);

iterator erase(iterator where);
iterator erase(iterator first, iterator last);
size_type erase(const Key& keyval);
void clear();
```

```
    void swap(unordered_set& right);

    const_iterator find(const Key& keyval) const;
    size_type count(const Key& keyval) const;
    std::pair<iterator, iterator>
        equal_range(const Key& keyval);
    std::pair<const_iterator, const_iterator>
        equal_range(const Key& keyval) const;
    };
```

Constructor:

unordered_set::unordered_set:

```
unordered_set {
        const unordered_set& right);
    explicit unordered_set(
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());
    template<class InIt>
        unordered_set(
        InIt first, InIt last,
        size_type nbuckets = N0,
        const Hash& hfn = Hash(),
        const Pred& comp = Pred(),
        const Alloc& al = Alloc());
```

The first constructor specifies a copy of the sequence controlled by `right`. The second constructor specifies an empty controlled sequence. The third constructor inserts the sequence of element values `[first, last)`.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from `right`. Otherwise:
- the minimum number of buckets is the argument `nbuckets`, if present; otherwise it is a default value described here as the implementation-defined value `N0`.
- the hash function object is the argument `hfn`, if present; otherwise it is `Hash()`.
- the comparison function object is the argument `comp`, if present; otherwise it is `Pred()`.
- the allocator object is the argument `al`, if present; otherwise, it is `Alloc()`.

Types:

unordered_set::allocator_type:

```
typedef Alloc allocator_type;
```

The type is a synonym for the template parameter `Alloc`.

unordered_set::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

unordered_set::const_local_iterator:

```
typedef T5 const_local_iterator;
```

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type T5.

unordered_set::const_pointer:

```
typedef Alloc::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

unordered_set::const_reference:

```
typedef Alloc::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

unordered_set::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

unordered_set::hasher:

```
typedef Hash hasher;
```

The type is a synonym for the template parameter Hash.

unordered_set::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T0.

unordered_set::key_equal:

```
typedef Pred key_equal;
```

The type is a synonym for the template parameter Pred.

unordered_set::key_type:

```
typedef Key key_type;
```

The type is a synonym for the template parameter Key.

unordered_set::local_iterator:

```
typedef T4 local_iterator;
```

The type describes an object that can serve as a forward iterator for a bucket. It is described here as a synonym for the implementation-defined type T4.

unordered_set::pointer:

```
typedef Alloc::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

unordered_set::reference:

```
typedef Alloc::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

unordered_set::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

unordered_set::value_type:

```
typedef Key value_type;
```

The type describes an element of the controlled sequence.

Member functions:

unordered_set::begin:

```
iterator begin();
    const_iterator begin() const;
    local_iterator begin(size_type nbucket);
    const_local_iterator begin(size_type nbucket) const;
```

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket nbucket (or just beyond the end of an empty bucket).

unordered_set::bucket:

```
    size_type bucket(const Key& keyval) const;
```

The member function returns the bucket number currently corresponding to the key value keyval.

unordered_set::bucket_count:

```
    size_type bucket_count() const;
```

The member function returns the current number of buckets.

unordered_set::bucket_size:

```
    size_type bucket_size(size_type nbucket) const;
```

The member functions returns the size of bucket number nbucket.

unordered_set::clear:

```
void clear();
```

The member function calls erase(begin(), end()).

unordered_set::count:

```
size_type count(const Key& keyval) const;
```

The member function returns the number of elements in the range delimited by `equal_range(keyval)`.

unordered_set::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

unordered_set::end:

```
iterator end();
    const_iterator end() const;
    local_iterator end(size_type nbucket);
    const_local_iterator end(size_type nbucket) const;
```

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket `nbucket`.

unordered_set::equal_range:

```
std::pair<iterator, iterator>
  equal_range(const Key& keyval);
std::pair<const_iterator, const_iterator>
  equal_range(const Key& keyval) const;
```

The member function returns a pair of iterators `X` such that `[X.first, X.second)` delimits just those elements of the controlled sequence that have equivalent ordering with `keyval`. If no such elements exist, both iterators are `end()`.

unordered_set::erase:

```
iterator erase(iterator where);
iterator erase(iterator first, iterator last);
size_type erase(const Key& keyval);
```

The first member function removes the element of the controlled sequence pointed to by `where`. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes the elements in the rang delimited by `equal_range(keyval)`. It returns the number of elements it removes.

The member functions never throw an exception.

unordered_set::find:

```
const_iterator find(const Key& keyval) const;
```

The member function returns `equal_range(keyval).first`.

unordered_set::get_allocator:

```
Alloc get_allocator() const;
```

The member function returns the stored `allocator object`.

unordered_set::hash_function:

```
  Hash hash_function() const;
```

The member function returns the stored hash function object.

unordered_set::insert:

```
iterator insert(const value_type& val);
iterator insert(iterator where, const value_type& val);
template<class InIt>
    void insert(InIt first, InIt last);
```

The first member function inserts the element `val` in the controlled sequence, then returns the iterator that designates the inserted element. The second member function returns `insert(val)`, using `where` as a starting place within the controlled sequence to search for the insertion point. (Insertion can possibly occur somewhat faster, if the insertion point immediately precedes or follows `where`.) The third member function inserts the sequence of element values, for each `where` in the range `[first, last)`, by calling `insert(*where)`.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, the container is left in a stable but unspecified state and the exception is rethrown.

unordered_set::key_eq:

```
Pred key_eq() const;
```

The member function returns the stored comparison function object.

unordered_set::load_factor:

```
float load_factor() const;
```

The member function returns `(float)size() / (float)bucket_count()`, the average number of elements per bucket.

unordered_set::max_bucket_count:

```
size_type max_bucket_count() const;
```

The member function returns the maximum number of buckets currently permitted.

unordered_set::max_load_factor:

```
float max_load_factor() const;
void max_load_factor(float factor);
```

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with `factor`.

unordered_set::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

unordered_set::rehash:

```
void rehash(size_type nbuckets);
```

The member function alters the number of buckets to be at least `nbuckets` and rebuilds the hash table as needed.

unordered_set::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

unordered_set::swap:

```
void swap(unordered_set& right);
```

The member function swaps the controlled sequences between `*this` and `right`. If `get_allocator() == right.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored function object of type `Tr`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

<utility>

Description

Include the STL standard header **<utility>** to define several templates of general use throughout the Standard Template Library.

Note: Some classes and functions in <utility> have been added with TR1. To enable these, you must define the macro __IBMCPP_TR1__ .

Four template operators — `operator!=`, `operator<=`, `operator>`, and `operator>=` — define a **total ordering** on pairs of operands of the same type, given definitions of `operator==` and `operator<`.

If an implementation supports namespaces, these template operators are defined in the **rel_ops** namespace, nested within the `std` namespace. If you wish to make use of these template operators, write the declaration:

```
using namespace std::rel_ops;
```

which promotes the template operators into the current namespace.

Synopsis

```
namespace std {
template<class T, class Ty2>
    struct pair;

        // TEMPLATE FUNCTIONS
template<class Ty1, class Ty2>
    pair<Ty, Ty2> make_pair(Ty1 val1, Ty2 val2);
template<class Ty1, class Ty2>
    bool operator==(const pair<Ty, Ty2>& left,
        const pair<Ty1, Ty2>& right);
template<class Ty1, class Ty2>
    bool operator!=(const pairTy, Ty2>& left,
        const pair<Ty1, Ty2>& right);
template<class Ty1, class Ty2>
    bool operator<(const pair<Ty, Ty2>& left,
        const pair<Ty1, Ty2>& right);
template<class Ty1, class Ty2>
    bool operator>(const pair<Ty1, Ty2>& left,
        const pair<Ty1, Ty2>& right);
template<class Ty1, class Ty2>
    bool operator<=(const pair<Ty1, Ty2>& left,
        const pair<Ty1, Ty2>& right);
```

```
template<class Ty1, class Ty2>
    bool operator>=(const pair<Ty1, Ty2>& left,
        const pair<Ty1, Ty2>& right);

namespace rel_ops {
    template<class Ty>
        bool operator!=(const Ty& left, const Ty& right);
    template<class Ty>
        bool operator<=(const Ty& left, const Ty& right);
    template<class Ty>
        bool operator>(const Ty& left, const Ty& right);
    template<class Ty>
        bool operator>=(const Ty& left, const Ty& right);
        }; // namespace rel_ops

    namespace tr1 {  [added with (TR1)]
template<int Idx, class T1, class T2>
    RI& get(pair<T1, T2>& pr);
template<int Idx, class T1, class T2>
    const RI& get(const pair<T1, T2>& pr);

template<class T1, class T2>
    class tuple_element<0, pair<T1, T2> >;
template<class T1, class T2>
    class tuple_element<1, pair<T1, T2> >;

template<class T1, class T2>
    class tuple_size<pair<T1, T2> >;
    } // namespace tr1
 } // namespace std
```

Classes

pair

```
template<class Ty1, class Ty2>
    struct pair {
    typedef Ty1 first_type;
    typedef Ty2 second_type
    Ty1 first;
    Ty2 second;
    pair();
    pair(const Ty1& val1, const Ty2& val2);
    template<class Other1, class Other2>
        pair(const pair<Other1, Other2>& right);
    };
```

The template class stores a pair of objects, **first**, of type Ty1, and **second**, of type Ty2. The type definition **first_type**, is the same as the template parameter Ty1, while **second_type**, is the same as the template parameter Ty2.

The first default constructor initializes first to Ty1() and second to Ty2(). The second constructor initializes first to val1 and second to val2. The third template constructor initializes first to right.first and second to right.second. Ty1 and Ty2 each need supply only a default constructor, single-argument constructor, and a destructor.

tuple_element

```
template<class T1, class T2>
    class tuple_element<0, pair<T1, T2> > {
    typedef T1 type;
    };
```

```
template<class T1, class T2>
    class tuple_element<1, pair<T1, T2> > {
    typedef T2 type;
    };
```

[Added with TR1]

The templates are specializations of the template class "tuple_element" on page 445. Each has a nested typedef `type` that is a synonym for the type of the corresponding `pair` element.

tuple_size

```
template<class T1, class T2>
    class tuple_size<pair<T1, T2> > {
    static const unsigned value = 2;
    };
```

[Added with TR1]

The template is a specialization of the template class "tuple_size." It has a member `value` that is an integral constant expression whose value is 2.

Functions

get

```
template<int Idx, class T1, class T2>
    RI& get(pair<T1, T2>& pr);
template<int Idx, class T1, class T2>
    const RI& get(const pair<T1, T2>& pr);
```

[Added with TR1]

The template functions return a reference to an element of its `pair` argument. If the value of `Idx` is 0 the functions return `pr.first` and if the value of `Idx` is 1 the functions return `pr.second`. The type `RI` is the type of the returned element.

make_pair

```
template<class Ty1, class Ty2>
    pair<Ty1, Ty2> make_pair(Ty1 val1, Ty2 val2);
```

The template function returns `pair<Ty1, Ty2>(val1, val2)`.

operator!=

```
template<class Ty>
    bool operator!=(const Ty& left, const Ty& right);
template<class Ty1, class Ty2>
    bool operator!=(const pair<Ty1, Ty2>& left, const pair<Ty1, Ty2>& right);
```

The template function returns `!(left == right)`.

operator==

```
template<class Ty1, class Ty2>
    bool operator==(const pair<Ty1, Ty2>& left,
        const pair<Ty1, Ty2>& right);
```

The template function returns `left.first == right.first && left.second == right.second`.

operator<

```
template<class Ty1, class Ty2>
    bool operator<(const pair<Ty1, Ty2>& left,
        const pair<Ty1, Ty2>& right);
```

The template function returns `left.first < right.first || !(right.first < left.first) && left.second < right.second`.

operator<=

```
template<class Ty>
    bool operator<=(const Ty& left, const Ty& right);
template<class Ty1, class Ty2>
    bool operator<=(const pair<Ty1, Ty2>& left, const pair<Ty1, Ty2>& right);
```

The template function returns `!(right < left)`.

operator>

```
template<class Ty>
    bool operator>(const Ty& left, const Ty& right);
template<class Ty1, class Ty2>
    bool operator>(const pair<Ty1, Ty2>& left, const pair<Ty1, Ty2>& right);
```

The template function returns `right < left`.

operator>=

```
template<class Ty>
    bool operator>=(const Ty& left, const Ty& right);
template<class Ty1, class Ty2>
    bool operator>=(const pair<Ty1, Ty2>& left, const pair<Ty1, Ty2>& right);
```

The template function returns `!(left < right)`.

<valarray>

Description

Include the standard header **<valarray>** to define the template class valarray and numerous supporting template classes and functions. These template classes and functions are permitted unusual latitude, in the interest of improved performance. Specifically, any function returning `valarray<T>` may return an object of some other type `T'`. In that case, any function that accepts one or more arguments of type `valarray<T>` must have overloads that accept arbitrary combinations of those arguments, each replaced with an argument of type `T'`. (Put simply, the only way you can detect such a substitution is to go looking for it.)

Synopsis

```
namespace std {
class slice;
class gslice;

        // TEMPLATE CLASSES
template<class T>
    class valarray;
template<class T>
    class slice_array;
template<class T>
    class gslice_array;
template<class T>
    class mask_array;
```

```
template<class T>
    class indirect_array;

        // TEMPLATE FUNCTIONS
template<class T>
    valarray<T> operator*(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator*(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator*(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator/(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator/(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator/(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator%(const valarray<T>& x,
        const vararray<T>& y);
template<class T>
    valarray<T> operator%(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator%(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator+(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator+(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator+(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator-(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator-(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator-(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator^(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator^(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator^(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator&(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator&(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator&(const T& x,
        const valarray<T>& y);
```

```
template<class T>
    valarray<T> operator|(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator|(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator|(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator<<(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator<<(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator<<(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator>>(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator>>(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator>>(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator&&(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator&&(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator&&(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator||(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator||(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator||(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator==(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator==(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator==(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator!=(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator!=(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator!=(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator<(const valarray<T>& x,
        const valarray<T>& y);
```

```
template<class T>
    valarray<bool> operator<(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator<(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator>=(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator>=(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator>=(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator>(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator>(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator>(const T& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator<=(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator<=(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator<=(const T& x,
        const valarray<T>& y);

template<class T>
    valarray<T> abs(const valarray<T>& x);
template<class T>
    valarray<T> acos(const valarray<T>& x);
template<class T>
    valarray<T> asin(const valarray<T>& x);
template<class T>
    valarray<T> atan(const valarray<T>& x);
template<class T>
    valarray<T> atan2(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> atan2(const valarray<T> x, const T& y);
template<class T>
    valarray<T> atan2(const T& x, const valarray<T>& y);
template<class T>
    valarray<T> cos(const valarray<T>& x);
template<class T>
    valarray<T> cosh(const valarray<T>& x);
template<class T>
    valarray<T> exp(const valarray<T>& x);
template<class T>
    valarray<T> log(const valarray<T>& x);
template<class T>
    valarray&tt;T> log10(const valarray<T>& x);
template<class T>
    valarray<T> pow(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> pow(const valarray<T> x, const T& y);
template<class T>
    valarray<T> pow(const T& x, const valarray<T>& y);
template<class T>
    valarray<T> sin(const valarray<T>& x);
```

```
template<class T>
    valarray<T> sinh(const valarray<T>& x);
template<class T>
    valarray<T> sqrt(const valarray<T>& x);
template<class T>
    valarray<T> tan(const valarray<T>& x);
template<class T>
    valarray<T> tanh(const valarray<T>& x);
    }
```

Classes

gslice

Description: The class stores the parameters that characterize a gslice_array when an object of class `gslice` appears as a subscript for an object of class `valarray<T>`. The stored values include:

- a **starting index**
- a **length vector** of class `valarray<size_t>`
- a **stride vector** of class `valarray<size_t>`

The two vectors must have the same length.

Synopsis:

```
class gslice {
public:
    gslice();
    gslice(size_t st,
        const valarray<size_t> len,
        const valarray<size_t> str);
    size_t start() const;
    const valarray<size_t> size() const;
    const valarray<size_t> stride() const;
    };
```

Constructor:

gslice::gslice:

```
gslice();
gslice(size_t st,
    const valarray<size_t> len,
    const valarray<size_t> str);
```

The default constructor stores zero for the starting index, and zero-length vectors for the length and stride vectors. The second constructor stores `st` for the starting index, `len` for the length vector, and `str` for the stride vector.

Member functions:

gslice::size:

```
const valarray<size_t> size() const;
```

The member function returns the stored length vector.

gslice::start:

```
size_t start() const;
```

The member function returns the stored starting index.

gslice::stride:

```
const valarray<size_t> stride() const;
```

The member function returns the stored stride vector.

gslice_array

Description: The class describes an object that stores a reference to an object x of class valarray<T>, along with an object gs of class gslice which describes the sequence of elements to select from the valarray<T> object.

You construct a gslice_array<T> object only by writing an expression of the form x[gs]. The member functions of class gslice_array then behave like the corresponding function signatures defined for valarray<T>, except that only the sequence of selected elements is affected.

The sequence is determined as follows. For a length vector gs.size() of length N, construct the index vector valarray<size_t> idx(0, N). This designates the initial element of the sequence, whose index k within x is given by the mapping:

```
k = start;
for (size_t i = 0; i < gs.size()[i]; ++i)
    k += idx[i] * gs.stride()[i];
```

The successor to an index vector value is given by:

```
for (size_t i = N; 0 < i--; )
    if (++idx[i] < gs.size()[i])
        break;
    else
        idx[i] = 0;
```

For example:

```
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);
// x[gslice(3, len, str)] selects elements with
//   indices 3, 5, 7, 10, 12, 14
```

Synopsis:

```
template<class T>
    class gslice_array {
public:
    typedef T value_type;
    void operator=(const valarray<T> x) const;
    void operator=(const T& x);
    void operator*=(const valarray<T> x) const;
    void operator/=(const valarray<T> x) const;
    void operator%=(const valarray<T> x) const;
    void operator+=(const valarray<T> x) const;
    void operator-=(const valarray<T> x) const;
    void operator^=(const valarray<T> x) const;
    void operator&=(const valarray<T> x) const;
    void operator|=(const valarray<T> x) const;
    void operator<<=(const valarray<T> x) const;
    void operator>>=(const valarray<T> x) const;
private:
    void gslice_array();  // not defined
    void gslice_array(
        const gslice_array&);  // not defined
    gslice_array& operator=(
        const gslice_array&);  // not defined
    };
```

indirect_array

Description: The class describes an object that stores a reference to an object x of class valarray<T>, along with an object xa of class valarray<size_t> which describes the sequence of elements to select from the valarray<T> object.

You construct an indirect_array<T> object only by writing an expression of the form x[xa]. The member functions of class indirect_array then behave like the corresponding function signatures defined for valarray<T>, except that only the sequence of selected elements is affected.

The sequence consists of xa.size() elements, where element i becomes the index xa[i] within x. For example:

```
const size_t vi[] = {7, 5, 2, 3, 8};
// x[valarray<size_t>(vi, 5)] selects elements with
//   indices 7, 5, 2, 3, 8
```

Synopsis:

```
template<class T>
    class indirect_array {
public:
    typedef T value_type;
    void operator=(const valarray<T> x) const;
    void operator=(const T& x);
    void operator*=(const valarray<T> x) const;
    void operator/=(const valarray<T> x) const;
    void operator%=(const valarray<T> x) const;
    void operator+=(const valarray<T> x) const;
    void operator-=(const valarray<T> x) const;
    void operator^=(const valarray<T> x) const;
    void operator&=(const valarray<T> x) const;
    void operator|=(const valarray<T> x) const;
    void operator<<=(const valarray<T> x) const;
    void operator>>=(const valarray<T> x) const;
private:
private:
    void indirect_array();  // not defined
    void indirect_array(
        const indirect_array&);  // not defined
    indirect_array& operator=(
        const indirect_array&);  // not defined
    };
```

mask_array

Description: The class describes an object that stores a reference to an object x of class valarray<T>, along with an object ba of class valarray<bool> which describes the sequence of elements to select from the valarray<T> object.

You construct a mask_array<T> object only by writing an expression of the form x[xa]. The member functions of class mask_array then behave like the corresponding function signatures defined for valarray<T>, except that only the sequence of selected elements is affected.

The sequence consists of at most ba.size() elements. An element j is included only if ba[j] is true. Thus, there are as many elements in the sequence as there are true elements in ba. If i is the index of the lowest true element in ba, then x[i] is element zero in the selected sequence. For example:

```
const bool vb[] = {false, false, true, true, false, true};
// x[valarray<bool>(vb, 56] selects elements with
//   indices 2, 3, 5
```

Synopsis:

```
template<class T>
    class mask_array {
public:
    typedef T value_type;
    void operator=(const valarray<T> x) const;
    void operator=(const T& x);
    void operator*=(const valarray<T> x) const;
    void operator/=(const valarray<T> x) const;
    void operator%=(const valarray<T> x) const;
    void operator+=(const valarray<T> x) const;
    void operator-=(const valarray<T> x) const;
    void operator^=(const valarray<T> x) const;
    void operator&=(const valarray<T> x) const;
    void operator|=(const valarray<T> x) const;
    void operator<<=(const valarray<T> x) const;
    void operator>>=(const valarray<T> x) const;
private:
    void mask_array();  // not defined
    void mask_array(
        const mask_array&);  // not defined
    gslice_array& operator=(
        const mask_array&);  // not defined
    };
```

slice

Description: The class stores the parameters that characterize a slice_array when an object of class `slice` appears as a subscript for an object of class `valarray<T>`. The stored values include:

- a **starting index**
- a **total length**
- a **stride**, or distance between subsequent indices

Synopsis:

```
class slice {
public:
    slice();
    slice(size_t st, size_t len, size_t str);
    size_t start() const;
    size_t size() const;
    size_t stride() const;
    };
```

Constructor:

slice::slice:

```
slice();
slice(size_t st,
        const valarray<size_t> len, const valarray<size_t> str);
```

The default constructor stores zeros for the starting index, total length, and stride. The second constructor stores `st` for the starting index, `len` for the total length, and `str` for the stride.

Member functions:

slice::size:

```
size_t size() const;
```

The member function returns the stored total length.

slice::start:

```
size_t start() const;
```

The member function returns the stored starting index.

slice::stride:

```
size_t stride() const;
```

The member function returns the stored stride.

slice_array

Description: The class describes an object that stores a reference to an object `x` of class `valarray<T>`, along with an object `sl` of class slice which describes the sequence of elements to select from the `valarray<T>` object.

You construct a `slice_array<T>` object only by writing an expression of the form x[sl]. The member functions of class `slice_array` then behave like the corresponding function signatures defined for `valarray<T>`, except that only the sequence of selected elements is affected.

The sequence consists of `sl.size()` elements, where element `i` becomes the index `sl.start() + i * sl.stride()` within `x`. For example:

```
// x[slice(2, 5, 3)] selects elements with
//   indices 2, 5, 8, 11, 14
```

Synopsis:

```
template<class T>
    class slice_array {
public:
    typedef T value_type;
    void operator=(const valarray<T> x) const;
    void operator=(const T& x);
    void operator*=(const valarray<T> x) const;
    void operator/=(const valarray<T> x) const;
    void operator%=(const valarray<T> x) const;
    void operator+=(const valarray<T> x) const;
    void operator-=(const valarray<T> x) const;
    void operator^=(const valarray<T> x) const;
    void operator&=(const valarray<T> x) const;
    void operator|=(const valarray<T> x) const;
    void operator<<=(const valarray<T> x) const;
    void operator>>=(const valarray<T> x) const;
private:
    void slice_array();  // not defined
    void slice_array(
        const slice_array&);  // not defined
    slice_array& operator=(
        const slice_array&);  // not defined
    };
```

valarray

Description: The template class describes an object that controls a varying-length sequence of elements of type T. The sequence is stored as an array of T. It differs from template class vector in two important ways:

- It defines numerous arithmetic operations between corresponding elements of valarray<T> objects of the same type and length, such as x = cos(y) + sin(z).
- It defines a variety of interesting ways to subscript a valarray<T> object, by overloading operator[].

An object of class T:

- has a public default constructor, destructor, copy constructor, and assignment operator — with conventional behavior
- defines the arithmetic operators and math functions, as needed, that are defined for the floating-point types — with conventional behavior

In particular, no subtle differences may exist between copy construction and default construction followed by assignment. And none of the operations on objects of class T may throw exceptions.

Synopsis:

```
template<class T>
    class valarray {
public:
    typedef T value_type;
    valarray();
    explicit valarray(size_t n);
    valarray(const T& val, size_t n));
    valarray(const T *p, size_t n);
    valarray(const slice_array<T>& sa);
    valarray(const gslice_array<T>& ga);
    valarray(const mask_array<T>& ma);
    valarray(const indirect_array<T>& ia);
    valarray<T>& operator=(const valarray<T>& va);
    valarray<T>& operator=(const T& x);
    valarray<T>& operator=(const slice_array<T>& sa);
    valarray<T>& operator=(const gslice_array<T>& ga);
    valarray<T>& operator=(const mask_array<T>& ma);
    valarray<T>& operator=(const indirect_array<T>& ia);
    T operator[](size_t n) const;
    T& operator[](size_t n);
    valarray<T> operator[](slice sa) const;
    slice_array<T> operator[](slice sa);
    valarray<T> operator[](const gslice& ga) const;
    gslice_array<T> operator[](const gslice& ga);
    valarray<T>
        operator[](const valarray<bool>& ba) const;
    mask_array<T> operator[](const valarray<bool>& ba);
    valarray<T>
        operator[](const valarray<size_t>& xa) const;
    indirect_array<T>
        operator[](const valarray<size_t>& xa);
    valarray<T> operator+();
    valarray<T> operator-();
    valarray<T> operator~();
    valarray<bool> operator!();
    valarray<T>& operator*=(const valarray<T>& x);
    valarray<T>& operator*=(const T& x);
    valarray<T>& operator/=(const valarray<T>& x);
    valarray<T>& operator/=(const T& x);
    valarray<T>& operator%=(const valarray<T>& x);
    valarray<T>& operator%=(const T& x);
```

```
    valarray<T>& operator+=(const valarray<T>& x);
    valarray<T>& operator+=(const T& x);
    valarray<T>& operator-=(const valarray<T>& x);
    valarray<T>& operator-=(const T& x);
    valarray<T>& operator^=(const valarray<T>& x);
    valarray<T>& operator^=(const T& x);
    valarray<T>& operator&=(const valarray<T>& x);
    valarray<T>& operator&=(const T& x);
    valarray<T>& operator|=(const valarray<T>& x);
    valarray<T>& operator|=(const T& x);
    valarray<T>& operator<<=(const valarray<T>& x);
    valarray<T>& operator<<=(const T& x);
    valarray<T>& operator>>=(const valarray<T>& x);
    valarray<T>& operator>>=(const T& x);
    operator T *();
    operator const T *() const;
    size_t size() const;
    T sum() const;
    T max() const;
    T min() const;
    valarray<T> shift(int n) const;
    valarray<T> cshift(int n) const;
    valarray<T> apply(T fn(T)) const;
    valarray<T> apply(T fn(const T&)) const;
    void resize(size_t n);
    void resize(size_t n, const T& c);
   };
```

Constructor:

valarray::valarray:

```
valarray();
explicit valarray(size_t n);
valarray(const T& val, size_t n));
valarray(const T *p, size_t n);
valarray(const slice_array<T>& sa);
valarray(const gslice_array<T>& ga);
valarray(const mask_array<T>& ma);
valarray(const indirect_array<T>& ia);
```

The first (default) constructor initializes the object to an empty array. The next three constructors each initialize the object to an array of n elements as follows:

- For explicit valarray(size_t n), each element is initialized with the default constructor.
- For valarray(const T& val, size_t n)), each element is initialized with val.
- For valarray(const T *p, size_t n), the element at position I is initialized with p[I].

Each of the remaining constructors initializes the object to a valarray<T> object determined by the argument.

Types:

valarray::value_type:

```
typedef T value_type;
```

The type is a synonym for the template parameter T.

Member functions:

valarray::apply:

```
valarray<T> apply(T fn(T)) const;
valarray<T> apply(T fn(const T&)) const;
```

The member function returns an object of class valarray<T>, of length size(), each of whose elements I is fn((*this)[I]).

valarray::cshift:

```
valarray<T> cshift(int n) const;
```

The member function returns an object of class valarray<T>, of length size(), each of whose elements I is (*this)[(I + n) % size()]. Thus, if element zero is taken as the leftmost element, a positive value of n shifts the elements circularly left n places.

valarray::max:

```
T max() const;
```

The member function returns the value of the largest element of *this, which must have nonzero length. If the length is greater than one, it compares values by applying operator< between pairs of corresponding elements of class T.

valarray::min:

```
T min() const;
```

The member function returns the value of the smallest element of *this, which must have nonzero length. If the length is greater than one, it compares values by applying operator< between pairs of elements of class T.

*valarray::operator T *:*

```
operator T *();
operator const T *() const;
```

Both member functions return a pointer to the first element of the controlled array, which must have at least one element.

valarray::resize:

```
void resize(size_t n);
void resize(size_t n, const T& c);
```

The member functions both ensure that size() henceforth returns n. If it must make the controlled sequence longer, the first member function appends elements with value T(), while the second member function appends elements with value c. To make the controlled sequence shorter, both member functions remove and delete elements with subscripts in the range [n, size()). Any pointers or references to elements in the controlled sequence are invalidated.

valarray::shift:

```
valarray<T> shift(int n) const;
```

The member function returns an object of class valarray<T>, of length size(), each of whose elements I is either (*this)[I + n], if I + n is a valid subscript, or T(). Thus, if element zero is taken as the leftmost element, a positive value of n shifts the elements left n places, with zero fill.

valarray::size:

```
size_t size() const;
```

The member function returns the number of elements in the array.

valarray::sum:

```
T sum() const;
```

The member function returns the sum of all elements of `*this`, which must have nonzero length. If the length is greater than one, it adds values to the sum by applying `operator+=` between pairs of elements of class `T`.

Operators:

valarray::operator!:

```
valarray<bool> operator!();
```

The member operator returns an object of class `valarray<bool>`, of length `size()`, each of whose elements `I` is `!(*this)`.

valarray::operator%=:

```
valarray<T>& operator%=(const valarray<T>& x);
valarray<T>& operator%=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] % x[I]`. It returns `*this`.

valarray::operator&=:

```
valarray<T>& operator&=(const valarray<T>& x);
valarray<T>& operator&=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] & x[I]`. It returns `*this`.

valarray::operator>>=:

```
valarray<T>& operator>>=(const valarray<T>& x);
valarray<T>& operator>>=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I]> > x[I]`. It returns `*this`.

valarray::operator<<=:

```
valarray<T>& operator<<=(const valarray<T>& x);
valarray<T>& operator<<=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] << x[I]`. It returns `*this`.

valarray::operator=:*

```
valarray<T>& operator*=(const valarray<T>& x);
valarray<T>& operator*=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] * x[I]`. It returns `*this`.

valarray::operator+:

```
valarray<T> operator+();
```

The member operator returns an object of class valarray<T>, of length size(), each of whose elements I is (*this)[I].

valarray::operator+=:

```
valarray<T>& operator+=(const valarray<T>& x);
valarray<T>& operator+=(const T& x);
```

The member operator replaces each element I of *this with (*this)[I] + x[I]. It returns *this.

valarray::operator-:

```
valarray<T> operator-();
```

The member operator returns an object of class valarray<T>, of length size(), each of whose elements I is -(*this)[I].

valarray::operator-=:

```
valarray<T>& operator-=(const valarray<T>& x);
valarray<T>& operator-=(const T& x);
```

The member operator replaces each element I of *this with (*this)[I] - x[I]. It returns *this.

valarray::operator/=:

```
valarray<T>& operator/=(const valarray<T>& x);
valarray<T>& operator/=(const T& x);
```

The member operator replaces each element I of *this with (*this)[I] / x[I]. It returns *this.

valarray::operator=:

```
valarray<T>& operator=(const valarray<T>& va);
    valarray<T>& operator=(const T& x);
valarray<T>& operator=(const slice_array<T>& sa);
valarray<T>& operator=(const gslice_array<T>& ga);
valarray<T>& operator=(const mask_array<T>& ma);
valarray<T>& operator=(const indirect_array<T>& ia);
```

The first member operator replaces the controlled sequence with a copy of the sequence controlled by va. The second member operator replaces each element of the controlled sequence with a copy of x. The remaining member operators replace those elements of the controlled sequence selected by their arguments, which are generated only by operator[]. If the value of a member in the replacement controlled sequence depends on a member in the initial controlled sequence, the result is undefined.

If the length of the controlled sequence changes, the result is generally undefined. In this implementation, however, the effect is merely to invalidate any pointers or references to elements in the controlled sequence.

valarray::operator[]:

```
T& operator[](size_t n);
slice_array<T> operator[](slice sa);
gslice_array<T> operator[](const gslice& ga);
mask_array<T> operator[](const valarray<bool>& ba);
indirect_array<T> operator[](const valarray<size_t>& xa);

T operator[](size_t n) const;
```

```
valarray<T> operator[](slice sa) const;
valarray<T> operator[](const gslice& ga) const;
valarray<T> operator[](const valarray<bool>& ba) const;
valarray<T> operator[](const valarray<size_t>& xa) const;
```

The member operator is overloaded to provide several ways to select sequences of elements from among those controlled by `*this`. The first group of five member operators work in conjunction with various overloads of operator= (and other assigning operators) to allow selective replacement slicing of the controlled sequence. The selected elements must exist.

The first member operator selects element `n`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
v0[3] = 'A';
// v0 == valarray<char>("abcAefghijklmnop", 16)
```

The second member operator selects those elements of the controlled sequence designated by `sa`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
v0[slice(2, 5, 3)] = v1;
// v0 == valarray<char>("abAdeBghCjkDmnEp", 16)
```

The third member operator selects those elements of the controlled sequence designated by `ga`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDEF", 6);
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);
v0[gslice(3, len, str)] = v1;
// v0 == valarray<char>("abcAeBgCijDlEnFp", 16)
```

The fourth member operator selects those elements of the controlled sequence designated by `ba`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABC", 3);
const bool vb[] = {false, false, true, true, false, true};
v0[valarray<bool>(vb, 6)] = v1;
// v0 == valarray<char>("abABeCghijklmnop", 16)
```

The fifth member operator selects those elements of the controlled sequence designated by `ia`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
const size_t vi[] = {7, 5, 2, 3, 8};
v0[valarray<size_t>(vi, 5)] = v1;
// v0 == valarray<char>("abCDeBgAEjklmnop", 16)
```

The second group of five member operators each construct an object that represents the value(s) selected. The selected elements must exist.

The sixth member operator returns the value of element `n`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
// v0[3] returns 'd'
```

The seventh member operator returns an object of class `valarray<T>` containing those elements of the controlled sequence designated by `sa`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
// v0[slice(2, 5, 3)] returns valarray<char>("cfilo", 5)
```

The eighth member operator selects those elements of the controlled sequence designated by ga. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);
// v0[gslice(3, len, str)] returns
//    valarray<char>("dfhkmo", 6)
```

The ninth member operator selects those elements of the controlled sequence designated by ba. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
const bool vb[] = {false, false, true, true, false, true};
// v0[valarray<bool>(vb, 6)] returns
//    valarray<char>("cdf", 3)
```

The last member operator selects those elements of the controlled sequence designated by ia. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
const size_t vi[] = {7, 5, 2, 3, 8};
// v0[valarray<size_t>(vi, 5)] returns
//    valarray<char>("hfcdi", 5)
```

valarray::operator^=:

```
valarray<T>& operator^=(const valarray<T>& x);
valarray<T>& operator^=(const T& x);
```

The member operator replaces each element I of *this with (*this)[I] ^ x[I]. It returns *this.

valarray::operator|=:

```
valarray<T>& operator|=(const valarray<T>& x);
valarray<T>& operator|=(const T& x);
```

The member operator replaces each element I of *this with (*this)[I] | x[I]. It returns *this.

valarray::operator~:

```
valarray<T> operator~();
```

The member operator returns an object of class valarray<T>, of length size(), each of whose elements I is ~(*this)[I].

Template functions

abs

```
template<class T>
    valarray<T> abs(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the absolute value of x[I].

acos

```
template<class T>
    valarray<T> acos(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the arccosine of x[I].

asin

```
template<class T>
    valarray<T> asin(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the arcsine of x[I].

atan

```
template<class T>
    valarray<T> atan(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the arctangent of x[I].

atan2

```
template<class T>
    valarray<T> atan2(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> atan2(const valarray<T> x, const T& y);
template<class T>
    valarray<T> atan2(const T& x, const valarray<T>& y);
```

The first template function returns an object of class valarray<T>, each of whose elements I is the arctangent of x[I] / y[I]. The second template function stores in element I the arctangent of x[I] / y. The third template function stores in element I the arctangent of x / y[I].

cos

```
template<class T>
    valarray<T> cos(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the cosine of x[I].

cosh

```
template<class T>
    valarray<T> cosh(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the hyperbolic cosine of x[I].

exp

```
template<class T>
    valarray<T> exp(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the exponential of x[I].

log

```
template<class T>
    valarray<T> log(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the natural logarithm of x[I].

log10

```
template<class T>
    valarray<T> log10(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the base-10 logarithm of `x[I]`.

pow

```
template<class T>
    valarray<T> pow(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> pow(const valarray<T> x, const T& y);
template<class T>
    valarray<T> pow(const T& x, const valarray<T>& y);
```

The first template function returns an object of class `valarray<T>`, each of whose elements `I` is `x[I]` raised to the `y[I]` power. The second template function stores in element `I` `x[I]` raised to the `y` power. The third template function stores in element `I` `x` raised to the `y[I]` power.

sin

```
template<class T>
    valarray<T> sin(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the sine of `x[I]`.

sinh

```
template<class T>
    valarray<T> sinh(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the hyperbolic sine of `x[I]`.

sqrt

```
template<class T>
    valarray<T> sqrt(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the square root of `x[I]`.

tan

```
template<class T>
    valarray<T> tan(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the tangent of `x[I]`.

tanh

```
template<class T>
    valarray<T> tanh(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the hyperbolic tangent of `x[I]`.

Types

valarray<bool>

```
class valarray<bool>
```

The type is a specialization of template class valarray, for elements of type `bool`.

Operators

operator!=

```
template<class T>
    valarray<bool> operator!=(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator!=(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator!=(const T& x,
       const valarray<T>& y);
```

The first template operator returns an object of class valarray<bool>, each of whose elements `I` is `x[I] != y[I]`. The second template operator stores in element `I` `x[I] != y`. The third template operator stores in element `I` `x != y[I]`.

operator%

```
template<class T>
    valarray<T> operator%(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator%(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator%(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] % y[I]`. The second template operator stores in element `I` `x[I] % y`. The third template operator stores in element `I` `x % y[I]`.

operator&

```
template<class T>
    valarray<T> operator&(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator&(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator&(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] & y[I]`. The second template operator stores in element `I` `x[I] & y`. The third template operator stores in element `I` `x & y[I]`.

operator&&

```
template<class T>
    valarray<bool> operator&&(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator&&(const valarray<T> x,
```

```
        const T& y);
template<class T>
    valarray<bool> operator&&(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class `valarray<bool>`, each of whose elements `I` is `x[I] && y[I]`. The second template operator stores in element `I` `x[I] && y`. The third template operator stores in element `I` `x && y[I]`.

operator>

```
template<class T>
    valarray<bool> operator>(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator>(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator>(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class valarray<bool>, each of whose elements `I` is `x[I] > y[I]`. The second template operator stores in element `I` `x[I] > y`. The third template operator stores in element `I` `x > y[I]`.

operator>>

```
template<class T>
    valarray<T> operator>>(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator>>(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator>>(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] >> y[I]`. The second template operator stores in element `I` `x[I] >> y`. The third template operator stores in element `I` `x >> y[I]`.

operator>=

```
template<class T>
    valarray<bool> operator>=(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator>=(const valarray<T> x, const T& y);
template<class T>
    valarray<bool> operator>=(const T& x, const valarray<T>& y);
```

The first template operator returns an object of class valarray<bool>, each of whose elements `I` is `x[I] >= y[I]`. The second template operator stores in element `I` `x[I] >= y`. The third template operator stores in element `I` `x >= y[I]`.

operator<

```
template<class T>
    valarray<bool> operator<(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator<(const valarray<T> x, const T& y);
template<class T>
    valarray<bool> operator<(const T& x, const valarray<T>& y);
```

The first template operator returns an object of class valarray<bool>, each of whose elements I is x[I] < y[I]. The second template operator stores in element I x[I] < y. The third template operator stores in element I x < y[I].

operator<<

```
template<class T>
    valarray<T> operator<<(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator<<(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator<<(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class valarray<T>, each of whose elements I is x[I] << y[I]. The second template operator stores in element I x[I] << y. The third template operator stores in element I x << y[I].

operator<=

```
template<class T>
    valarray<bool> operator<=(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator<=(const valarray<T> x, const T& y);
template<class T>
    valarray<bool> operator<=(const T& x, const valarray<T>& y);
```

The first template operator retrrns an object of class valarray<bool>, each of whose elements I is x[I] <= y[I]. The second template operator stores in element I x[I] <= y. The third template operator stores in element I x <= y[I].

operator*

```
template<class T>
    valarray<T> operator*(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator*(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator*(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class valarray<T>, each of whose elements I is x[I] * y[I]. The second template operator stores in element I x[I] * y. The third template operator stores in element I x * y[I].

operator+

```
template<class T>
    valarray<T> operator+(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator+(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator+(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class valarray<T>, each of whose elements I is x[I] + y[I]. The second template operator stores in element I x[I] + y. The third template operator stores in element I x + y[I].

operator-

```
template<class T>
    valarray<T> operator-(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator-(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator-(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] - y[I]`. The second template operator stores in element `I` `x[I] - y`. The third template operator stores in element `I` `x - y[I]`.

operator/

```
template<class T>
    valarray<T> operator/(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator/(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator/(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] / y[I]`. The second template operator stores in element `I` `x[I] / y`. The third template operator stores in element `I` `x / y[I]`.

operator==

```
template<class T>
    valarray<bool> operator==(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator==(const valarray<T> x, const T& y);
template<class T>
    valarray<bool> operator==(const T& x  const valarray<T>& y);
```

The first template operator returns an object of class valarray<bool>, each of whose elements `I` is `x[I] == y[I]`. The second template operator stores in element `I` `x[I] == y`. The third template operator stores in element `I` `x == y[I]`.

operator^

```
template<class T>
    valarray<T> operator^(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator^(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator^(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] ^ y[I]`. The second template operator stores in element `I` `x[I] ^ y`. The third template operator stores in element `I` `x ^ y[I]`.

operator|

```
template<class T>
    valarray<T> operator|(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> operator|(const valarray<T> x,
        const T& y);
template<class T>
    valarray<T> operator|(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] | y[I]`. The second template operator stores in element `I` `x[I] | y`. The third template operator stores in element `I` `x | y[I]`.

operator||

```
template<class T>
    valarray<bool> operator||(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator||(const valarray<T> x,
        const T& y);
template<class T>
    valarray<bool> operator||(const T& x,
        const valarray<T>& y);
```

The first template operator returns an object of class `valarray<bool>`, each of whose elements `I` is `x[I] || y[I]`. The second template operator stores in element `I` `x[I] || y`. The third template operator stores in element `I` `x || y[I]`.

<vector>

Description

Include the STL standard header **<vector>** to define the container template class `vector` and several supporting templates.

Synopsis

```
namespace std {
template<class T, class A>
    class vector;
template<class A>
    class vector<bool>;

        // TEMPLATE FUNCTIONS
template<class T, class A>
    bool operator==(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator!=(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator<(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
```

```
    bool operator<=(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    void swap(
        vector<T, A>& lhs,
        vector<T, A>& rhs);
    }
```

Macros

__IBM_FAST_VECTOR

```
#define __IBM_FAST_VECTOR 1
```

The __IBM_FAST_VECTOR macro defines a different iterator for the std::vector<> template class. This iterator results in faster code, but is not compatible with code using the default iterator for a std::vector<> template class. All uses of std::vector<> for a data type must use the same iterator. Add `-D__IBM_FAST_VECTOR` to the compile line, or `#define __IBM_FAST_VECTOR 1` to your source code to use the faster iterator for the std::vector<> template class. You must compile all sources with this macro.

Classes

vector

Description: The template class describes an object that controls a varying-length sequence of elements of type `T`. The sequence is stored as an array of `T`.

The object allocates and frees storage for the sequence it controls through a stored allocator object of class `A`. Such an allocator object must have the same external interface as an object of template class allocator. Note that the stored allocator object is *not* copied when the container object is assigned.

Vector reallocation occurs when a member function must grow the controlled sequence beyond its current storage capacity. Other insertions and erasures may alter various storage addresses within the sequence. In all such cases, iterators or references that point at altered portions of the controlled sequence become **invalid**.

Synopsis:

```
template<class T, class A = allocator<T> >
    class vector {
public:
    typedef A allocator_type;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer
        const_pointer;
    typedef typename A::reference reference;
    typedef typename A::const_reference
        const_reference;
    typedef typename A::value_type value_type;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
```

```
        const_reverse_iterator;
    typedef reverse_iterator<iterator>
        reverse_iterator;
    vector();
    explicit vector(const A& al);
    explicit vector(size_type n);
    vector(size_type n, const T& x);
    vector(size_type n, const T& x,
        const A& al);
    vector(const vector& x);
    template<class InIt>
        vector(InIt first, InIt last);
    template<class InIt>
        vector(InIt first, InIt last,
            const A& al);
    void reserve(size_type n);
    size_type capacity() const;
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    void resize(size_type n);
    void resize(size_type n, T x);
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    reference at(size_type pos);
    const_reference at(size_type pos) const;
    reference operator[](size_type pos);
    const_reference operator[](size_type pos);
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
    void push_back(const T& x);
    void pop_back();
    template<class InIt>
        void assign(InIt first, InIt last);
    void assign(size_type n, const T& x);
    iterator insert(iterator it, const T& x);
    void insert(iterator it, size_type n, const T& x);
    template<class InIt>
        void insert(iterator it, InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    void clear();
    void swap>(vector& x);
    };
```

Constructor:

vector::vector:

```
vector();
explicit vector(const A& al);
explicit vector(size_type n);
vector(size_type n, const T& x);
vector(size_type n, const T& x, const A& al);
vector(const vector& x);
template<class InIt>
```

```
    vector(InIt first, InIt last);
template<class InIt>
    vector(InIt first, InIt last, const A& al);
```

All constructors store an allocator object and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

The first two constructors specify an empty initial controlled sequence. The third constructor specifies a repetition of `n` elements of value `T()`. The fourth and fifth constructors specify a repetition of `n` elements of value `x`. The sixth constructor specifies a copy of the sequence controlled by `x`. If `InIt` is an integer type, the last two constructors specify a repetition of `(size_type)first` elements of value `(T)last`. Otherwise, the last two constructors specify the sequence `[first, last)`.

All constructors copy `N` elements and perform no interim reallocation.

Types:

vector::allocator_type:

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

vector::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

vector::const_pointer:

```
typedef typename A::const_pointer
    const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

vector::const_reference:

```
typedef typename A::const_reference
    const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

vector::const_reverse_iterator:

```
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse iterator for the controlled sequence.

vector::difference_type:

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type T3.

vector::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type T0.

vector::pointer:

```
typedef typename A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

vector::reference:

```
typedef typename A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

vector::reverse_iterator:

```
typedef reverse_iterator<iterator>
    reverse_iterator;
```

The type describes an object that can serve as a reverse iterator for the controlled sequence.

vector::size_type:

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type T2.

vector::value_type:

```
typedef typename A::value_type value_type;
```

The type is a synonym for the template parameter T.

Member functions:

vector::assign:

```
template<class InIt>
    void assign(InIt first, InIt last);
void assign(size_type n, const T& x);
```

If InIt is an integer type, the first member function behaves the same as assign((size_type)first, (T)last). Otherwise, the first member function replaces the sequence controlled by *this with the sequence [first, last), which must *not* overlap the initial controlled sequence. The second member function replaces the sequence controlled by *this with a repetition of n elements of value x.

vector::at:

```
const_reference at(size_type pos) const;
reference at(size_type pos);
```

The member function returns a reference to the element of the controlled sequence at position pos. If that position is invalid, the function throws an object of class out_of_range.

vector::back:

```
reference back();
const_reference back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

vector::begin:

```
const_iterator begin() const;
iterator begin();
```

The member function returns a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

vector::capacity:

```
size_type capacity() const;
```

The member function returns the storage currently allocated to hold the controlled sequence, a value at least as large as size().

vector::clear:

```
void clear();
```

The member function calls erase(begin(), end()).

vector::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

vector::end:

```
const_iterator end() const;
iterator end();
```

The member function returns a random-access iterator that points just beyond the end of the sequence.

vector::erase:

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by it. The second member function removes the elements of the controlled sequence in the range [first, last). Both return an iterator that designates the first element remaining beyond any elements removed, or end() if no such element exists.

Erasing N elements causes N destructor calls and an assignment for each of the elements between the insertion point and the end of the sequence. No reallocation occurs, so iterators and references become invalid only from the first element erased through the end of the sequence.

The member functions never throw an exception.

vector::front:

```
reference front();
const_reference front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

vector::get_allocator:

```
A get_allocator() const;
```

The member function returns the stored allocator object.

vector::insert:

```
iterator insert(iterator it, const T& x);
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
```

Each of the member functions inserts, before the element pointed to by `it` in the controlled sequence, a sequence specified by the remaining operands. The first member function inserts a single element with value `x` and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of `n` elements of value `x`.

If `InIt` is an integer type, the last member function behaves the same as `insert(it, (size_type)first, (T)last)`. Otherwise, the last member function inserts the sequence `[first, last)`, which must *not* overlap the initial controlled sequence.

When inserting a single element, the number of element copies is linear in the number of elements between the insertion point and the end of the sequence. When inserting a single element at the end of the sequence, the amortized number of element copies is constant. When inserting N elements, the number of element copies is linear in N plus the number of elements between the insertion point and the end of the sequence — except when the template member is specialized for `InIt` an input iterator, which behaves like N single insertions.

If reallocation occurs, the size of the controlled sequence at least doubles, and all iterators and references become invalid. If no reallocation occurs, iterators become invalid only from the point of insertion through the end of the sequence.

If an exception is thrown during the insertion of a single element, the container is left unaltered and the exception is rethrown. If an exception is thrown during the insertion of multiple elements, and the exception is not thrown while copying an element, the container is left unaltered and the exception is rethrown.

vector::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

vector::operator[]:

```
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
```

The member function returns a reference to the element of the controlled sequence at position pos. If that position is invalid, the behavior is undefined.

vector::pop_back:

```
void pop_back();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

The member function never throws an exception.

vector::push_back:

```
void push_back(const T& x);
```

The member function inserts an element with value x at the end of the controlled sequence.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

vector::rbegin:

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

vector::rend:

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

vector::reserve:

```
void reserve(size_type n);
```

If n is greater than max_size(), the member function reports a **length error** by throwing an object of class length_error. Otherwise, it ensures that capacity() henceforth returns at least n.

vector::resize:

```
void resize(size_type n);
void resize(size_type n, T x);
```

The member functions both ensure that size() henceforth returns n. If it must make the controlled sequence longer, the first member function appends elements with value `T()`, while the second member function appends elements with value `x`. To make the controlled sequence shorter, both member functions call `erase(begin() + n, end())`.

vector::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

vector::swap:

```
void swap(vector& x);
```

The member function swaps the controlled sequences between `*this` and `x`. If `get_allocator() == x.get_allocator()`, it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

vector<bool, A>

Description: The class is a partial specialization of template class vector for elements of type `bool`. It alters the definition of four member types (to optimize the packing and unpacking of elements) and adds two member functions. Its behavior is otherwise the same as for template class `vector`.

Synopsis:

```
template<class A>
    class vector<bool, A> {
public:
    class reference;
    typedef bool const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T4 pointer;
    typedef T5 const_pointer;
    void flip();
    static void swap(reference x, reference y);
// rest same as template class vector
    };
```

Types:

vector<bool, A>::const_iterator:

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T1`.

vector<bool, A>::const_pointer:

```
typedef T5 const_pointer;
```

The type describes an object that can serve as a pointer to a constant element of the controlled sequence. It is described here as a synonym for the unspecified type `T5`.

vector<bool, A>::const_reference:

```
typedef bool const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence, in this case bool.

vector<bool, A>::iterator:

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the unspecified type T0.

vector<bool, A>::pointer:

```
typedef T4 pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence. It is described here as a synonym for the unspecified type T4.

vector<bool, A>::reference:

```
class reference {
public:
    reference& operator=(const reference& x);
    reference& operator=(bool x);
    void flip();
    bool operator~() const;
    operator bool() const;
    };
```

The type describes an object that can serve as a reference to an element of the controlled sequence. Specifically, for two objects x and y of class reference:

- **bool(x)** yields the value of the element designated by x
- **~x** yields the inverted value of the element designated by x
- **x.flip()** inverts the value stored in x
- **y = bool(x)** and **y = x** both assign the value of the element designated by x to the element designated by y

It is unspecified how member functions of class vector<bool> construct objects of class reference that designate elements of a controlled sequence. The default constructor for class reference generates an object that refers to no such element.

Member functions:

vector<bool, A>::flip:

```
void flip();
```

The member function inverts the values of all the members of the controlled sequence.

vector<bool, A>::swap:

```
void swap(reference x, reference y);
```

The static member function swaps the members of the controlled sequences designated by x and y.

Template functions

operator!=

```
template<class T, class A>
    bool operator!=(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class A>
    bool operator==(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function overloads `operator==` to compare two objects of template class vector. The function returns `lhs.size () == rhs.size() && equal(lhs. begin(), lhs. end(), rhs.begin())`.

operator<

```
template<class T, class A>
    bool operator<(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `vector`. The function returns `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class T, class A>
    bool operator<=(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class A>
    bool operator>(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class A>
    bool operator>=(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class T, class A>
    void swap(
        vector <T, A>& lhs,
        vector <T, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

Appendix. C++ Library Supplementary Documentation

Multibyte Characters

A source character set or target character set can also contain **multibyte characters** (sequences of one or more bytes). Each sequence represents a single character in the **extended character set**. You use multibyte characters to represent large sets of characters, such as Kanji. A multibyte character can be a one-byte sequence that is a character from the basic C character set, an additional one-byte sequence that is implementation defined, or an additional sequence of two or more bytes that is implementation defined.

Any multibyte encoding that contains sequences of two or more bytes depends, for its interpretation between bytes, on a **conversion state** determined by bytes earlier in the sequence of characters. In the **initial conversion state** if the byte immediately following matches one of the characters in the basic C character set, the byte must represent that character.

For example, the **EUC encoding** is a superset of ASCII. A byte value in the interval [0xA1, 0xFE] is the first of a two-byte sequence (whose second byte value is in the interval [0x80, 0xFF]). All other byte values are one-byte sequences. Since all members of the basic C character set have byte values in the range [0x00, 0x7F] in ASCII, EUC meets the requirements for a multibyte encoding in Standard C. Such a sequence is *not* in the initial conversion state immediately after a byte value in the interval [0xA1, 0xFe]. It is ill-formed if a second byte value is not in the interval [0x80, 0xFF].

Multibyte characters can also have a **state-dependent encoding**. How you interpret a byte in such an encoding depends on a conversion state that involves both a **parse state**, as before, and a **shift state**, determined by bytes earlier in the sequence of characters. The **initial shift state**, at the beginning of a new multibyte character, is also the initial conversion state. A subsequent **shift sequence** can determine an **alternate shift state**, after which all byte sequences (including one-byte sequences) can have a different interpretation. A byte containing the value zero, however, always represents the null character. It cannot occur as any of the bytes of another multibyte character.

For example, the **JIS encoding** is another superset of ASCII. In the initial shift state, each byte represents a single character, except for two three-byte shift sequences:

- The three-byte sequence `"\x1B$B"` shifts to two-byte mode. Subsequently, two successive bytes (both with values in the range [0x21, 0x7E]) constitute a single multibyte character.
- The three-byte sequence `"\x1B(B"` shifts back to the initial shift state.

JIS also meets the requirements for a multibyte encoding in Standard C. Such a sequence is *not* in the initial conversion state when partway through a three-byte shift sequence or when in two-byte mode.

(Amendment 1 adds the type mbstate_t, which describes an object that can store a conversion state. It also relaxes the above rules for generalized multibyte characters, which describe the encoding rules for a broad range of wide streams.)

You can write multibyte characters in C source text as part of a comment, a character constant, a string literal, or a filename in an *include* directive. How such characters print is implementation defined. Each sequence of multibyte characters that you write must begin and end in the initial shift state. The program can also include multibyte characters in null-terminated C strings used by several library functions, including the format strings for printf and scanf. Each such character string must begin and end in the initial shift state.

Wide-Character Encoding

Each character in the extended character set also has an integer representation, called a **wide-character encoding**. Each extended character has a unique wide-character value. The value zero always corresponds to the **null wide character**. The type definition wchar_t specifies the integer type that represents wide characters.

You write a **wide-character constant** as `L'mbc'`, where `mbc` represents a single multibyte character. You write a **wide-character string literal** as `L"mbs"`, where `mbs` represents a sequence of zero or more multibyte characters. The wide-character string literal `L"xyz"` becomes a sequence of wide-character constants stored in successive bytes of memory, followed by a null wide character: `{L'x', L'y', L'z', L'\0'}`

The following library functions help you convert between the multibyte and wide-character representations of extended characters: btowc, mblen, mbrlen, mbrtowc, mbsrtowcs, mbstowcs, mbtowc, wcrtomb, wcsrtombs, wcstombs, wctob, and wctomb.

The macro MB_LEN_MAX specifies the length of the longest possible multibyte sequence required to represent a single character defined by the implementation across supported locales. And the macro MB_CUR_MAX specifies the length of the longest possible multibyte sequence required to represent a single character defined for the current locale.

For example, the string literal `"hello"` becomes an array of six *char*:

```
{'h', 'e', 'l', 'l', 'o', 0}
```

while the wide-character string literal `L"hello"` becomes an array of six integers of type wchar_t:

```
{L'h', L'e', L'l', L'l', L'o', 0}
```

Files and Streams

Text and Binary Streams Byte and Wide Streams Controlling Streams Stream States

A program communicates with the target environment by reading and writing **files** (ordered sequences of bytes). A file can be, for example, a data set that you can read and write repeatedly (such as a disk file), a stream of bytes generated by a program (such as a pipeline), or a stream of bytes received from or sent to a peripheral device (such as the keyboard or display). The latter two are **interactive files**. Files are typically the principal means by which to interact with a program.

You manipulate all these kinds of files in much the same way — by calling library functions. You include the standard header `<stdio.h>` to declare most of these functions.

Before you can perform many of the operations on a file, the file must be **opened**. Opening a file associates it with a **stream**, a data structure within the Standard C library that glosses over many differences among files of various kinds. The library maintains the state of each stream in an object of type **FILE**.

The target environment opens three files prior to program startup. You can open a file by calling the library function fopen with two arguments. The first argument is a filename, a multibyte string that the target environment uses to identify which file you want to read or write. The second argument is a C string that specifies:

- whether you intend to read data from the file or write data to it or both
- whether you intend to generate new contents for the file (or create a file if it did not previously exist) or leave the existing contents in place
- whether writes to a file can alter existing contents or should only append bytes at the end of the file
- whether you want to manipulate a text stream or a binary stream

Once the file is successfully opened, you can then determine whether the stream is **byte oriented** (a **byte stream**) or **wide oriented** (a **wide stream**). Wide-oriented streams are supported only with Amendment 1. A stream is initially **unbound**. Calling certain functions to operate on the stream makes it byte oriented, while certain other functions make it wide oriented. Once established, a stream maintains its orientation until it is closed by a call to `fclose` or `freopen`.

Text and Binary Streams

A **text stream** consists of one or more **lines** of text that can be written to a text-oriented display so that they can be read. When reading from a text stream, the program reads an `NL` (newline) at the end of each line. When writing to a text stream, the program writes an `NL` to signal the end of a line. To match differing conventions among target environments for representing text in files, the library functions can alter the number and representations of characters transmitted between the program and a text stream.

Thus, positioning within a text stream is limited. You can obtain the current file-position indicator by calling fgetpos or `ftell`. You can position a text stream at a position obtained this way, or at the beginning or end of the stream, by calling `fsetpos` or `fseek`. Any other change of position might well be not supported.

For maximum portability, the program should not write:

- empty files
- `space` characters at the end of a line
- partial lines (by omitting the `NL` at the end of a file)
- characters other than the printable characters, `NL`, and `HT` (horizontal tab)

If you follow these rules, the sequence of characters you read from a text stream (either as byte or multibyte characters) will match the sequence of characters you wrote to the text stream when you created the file. Otherwise, the library functions can remove a file you create if the file is empty when you close it. Or they can alter or delete characters you write to the file.

A **binary stream** consists of one or more bytes of arbitrary information. You can write the value stored in an arbitrary object to a (byte-oriented) binary stream and read exactly what was stored in the object when you wrote it. The library functions do not alter the bytes you transmit between the program and a binary stream.

They can, however, append an arbitrary number of null bytes to the file that you write with a binary stream. The program must deal with these additional null bytes at the end of any binary stream.

Thus, positioning within a binary stream is well defined, except for positioning relative to the end of the stream. You can obtain and alter the current file-position indicator the same as for a text stream. Moreover, the offsets used by `ftell` and `fseek` count bytes from the beginning of the stream (which is byte zero), so integer arithmetic on these offsets yields predictable results.

A **byte stream** treats a file as a sequence of bytes. Within the program, the stream looks like the same sequence of bytes, except for the possible alterations described above.

Byte and Wide Streams

While a **byte stream** treats a file as a sequence of bytes, a **wide stream** treats a file as a sequence of **generalized multibyte characters**, which can have a broad range of encoding rules. (Text and binary files are still read and written as described above.) Within the program, the stream looks like the corresponding sequence of wide characters. Conversions between the two representations occur within the Standard C library. The conversion rules can, in principle, be altered by a call to `setlocale` that alters the category LC_CTYPE. Each wide stream determines its conversion rules at the time it becomes wide oriented, and retains these rules even if the category LC_CTYPE subsequently changes.

Positioning within a wide stream suffers the same limitations as for text streams. Moreover, the file-position indicator may well have to deal with a state-dependent encoding. Typically, it includes both a byte offset within the stream and an object of type mbstate_t. Thus, the only reliable way to obtain a file position within a wide stream is by calling `fgetpos`, and the only reliable way to restore a position obtained this way is by calling `fsetpos`.

Controlling Streams

`fopen` returns the address of an object of type FILE. You use this address as the `stream` argument to several library functions to perform various operations on an open file. For a byte stream, all input takes place as if each character is read by calling `fgetc`, and all output takes place as if each character is written by calling `fputc`. For a wide stream (with Amendment 1), all input takes place as if each character is read by calling `fgetwc`, and all output takes place as if each character is written by calling `fputwc`.

You can **close** a file by calling `fclose`, after which the address of the FILE object is invalid.

A FILE object stores the state of a stream, including:

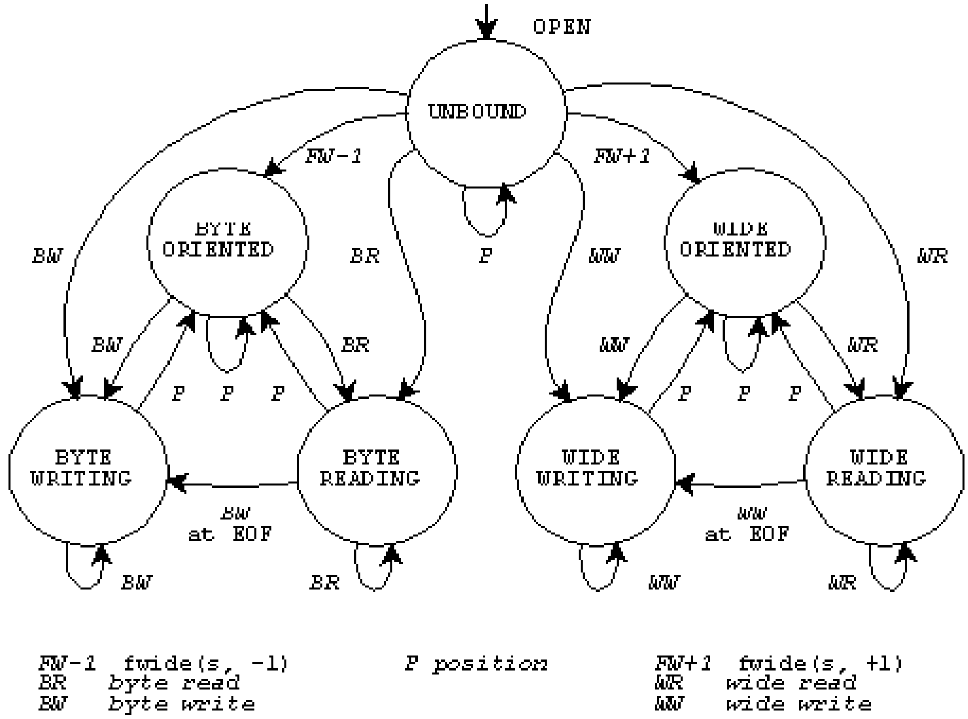
- an **error indicator —** set nonzero by a function that encounters a read or write error
- an **end-of-file indicator —** set nonzero by a function that encounters the end of the file while reading
- a **file-position indicator —** specifies the next byte in the stream to read or write, if the file can support positioning requests
- a **stream state —** specifies whether the stream will accept reads and/or writes and, with Amendment 1, whether the stream is unbound, byte oriented, or wide oriented

- a **conversion state** — remembers the state of any partly assembled or generated generalized multibyte character, as well as any shift state for the sequence of bytes in the file)
- a **file buffer** — specifies the address and size of an array object that library functions can use to improve the performance of read and write operations to the stream

Do not alter any value stored in a `FILE` object or in a file buffer that you specify for use with that object. You cannot copy a `FILE` object and portably use the address of the copy as a `stream` argument to a library function.

Stream States

The valid states, and state transitions, for a stream are shown in the diagram.



Each of the circles denotes a stable state. Each of the lines denotes a transition that can occur as the result of a function call that operates on the stream. Five groups of functions can cause state transitions.

Functions in the first three groups are declared in <stdio.h>:

- the **byte read functions** — `fgetc`, `fgets`, `fread`, `fscanf`, `getc`, `getchar`, `gets`, `scanf`, and `ungetc`
- the **byte write functions** — `fprintf`, `fputc`, `fputs`, `fwrite`, `printf`, `putc`, `putchar`, `puts`, `vfprintf`, and `vprintf`
- the **position functions** — `fflush`, `fseek`, `fsetpos`, and `rewind`

Functions in the remaining two groups are declared in <`wchar.h`>:

- the **wide read functions** — `fgetwc`, `fgetws`, `fwscanf`, `getwc`, `getwchar`, `ungetwc`, and `wscanf`
- the **wide write functions** — `fwprintf`, `fputwc`, `fputws`, `putwc`, `putwchar`, `vfwprintf`, `vwprintf`, and `wprintf`

For the stream `s`, the call `fwide(s, 0)` is always valid and never causes a change of state. Any other call to `fwide`, or to any of the five groups of functions described above, causes the state transition shown in the state diagram. If no such transition is shown, the function call is invalid.

The state diagram shows how to establish the orientation of a stream:
- The call `fwide(s, -1)`, or to a byte read or byte write function, establishes the stream as byte oriented.
- The call `fwide(s, 1)`, or to a wide read or wide write function, establishes the stream as wide oriented.

The state diagram shows that you must call one of the position functions between most write and read operations:
- You cannot call a read function if the last operation on the stream was a write.
- You cannot call a write function if the last operation on the stream was a read, unless that read operation set the end-of-file indicator.

Finally, the state diagram shows that a position operation never *decreases* the number of valid function calls that can follow.

Formatted Input

Scan Formats Scan Functions Scan Conversion Specifiers

Several library functions help you convert data values from text sequences that are generally readable by people to encoded internal representations. You provide a format string as the value of the `format` argument to each of these functions, hence the term **formatted input**. The functions fall into two categories:

The **byte scan functions** (declared in <stdio.h>) convert sequences of type *char* to internal representations, and help you scan such sequences that you read: `fscanf`, `scanf`, and `sscanf`. For these functions, a format string is a multibyte string that begins and ends in the initial shift state.

The **wide scan functions** (declared in <wchar.h> and hence added with **Amendment 1**) convert sequences of type wchar_t, to internal representations, and help you scan such sequences that you read: `fwscanf`, `wscanf` and `swscanf`. For these functions, a format string is a wide-character string. In the descriptions that follow, a wide character `wc` from a format string or a stream is compared to a specific (byte) character `c` as if by evaluating the expression `wctob(wc) == c`.

Scan Formats

A format string has the same general **syntax** for the scan functions as for the print functions: zero or more **conversion specifications**, interspersed with literal text and **white space**. For the scan functions, however, a conversion specification is one of the scan conversion specifications described below.

A scan function scans the format string once from beginning to end to determine what conversions to perform. Every scan function accepts a varying number of arguments, either directly or under control of an argument of type va_list. Some scan conversion specifications in the format string use the next argument in the list. A scan function uses each successive argument no more than once. Trailing arguments can be left unused.

In the description that follows, the integer conversions and floating-point conversions are the same as for the print functions.

Scan Functions

For the scan functions, literal text in a format string must match the next characters to scan in the input text. White space in a format string must match the longest possible sequence of the next zero or more white-space characters in the input. Except for the scan conversion specifier %n (which consumes no input), each **scan conversion specification** determines a pattern that one or more of the next characters in the input must match. And except for the scan conversion specifiers c, n, and [, every match begins by skipping any white space characters in the input.

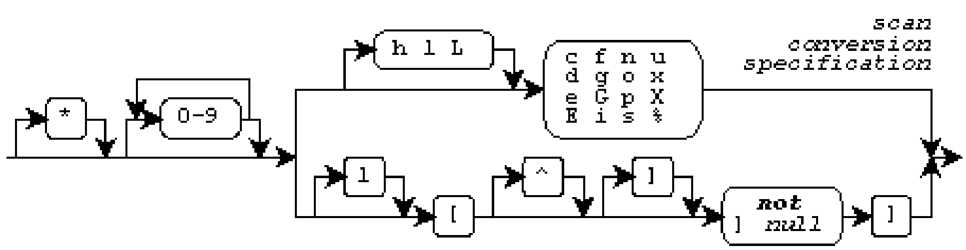
A scan function returns when:

- it reaches the terminating null in the format string
- it cannot obtain additional input characters to scan (**input failure**)
- a conversion fails (**matching failure**)

A scan function returns EOF if an input failure occurs before any conversion. Otherwise it returns the number of converted values stored. If one or more characters form a valid prefix but the conversion fails, the valid prefix is consumed before the scan function returns. Thus:

```
scanf("%i", &i)       consumes 0X from field 0XZ
scanf("%f", &f)       consumes 3.2E from field 3.2EZ
```

A scan conversion specification typically converts the matched input characters to a corresponding encoded value. The next argument value must be the address of an object. The conversion converts the encoded representation (as necessary) and stores its value in the object. A scan conversion specification has the format shown in the diagram.



Following the percent character (%) in the format string, you can write an asterisk (*) to indicate that the conversion should not store the converted value in an object.

Following any `*`, you can write a nonzero **field width** that specifies the maximum number of input characters to match for the conversion (not counting any white space that the pattern can first skip).

Scan Conversion Specifiers

Following any field width, you must write a one-character **scan conversion specifier**, either a one-character code or a scan set, possibly preceded by a one-character qualifier. Each combination determines the type required of the next argument (if any) and how the scan functions interpret the text sequence and converts it to an encoded value. The integer and floating-point conversions also determine what base to assume for the text representation. (The base is the `base` argument to the functions `strtol` and `strtoul`.) The following table lists all defined combinations and their properties.

Conversion Specifier	Argument Type	Conversion Function	Base
%c	char x[]		
%lc	wchar_t x[]		
%d	int *x	strtol	10
%hd	short *x	strtol	10
%ld	long *x	strtol	10
%e	float *x	strtod	10
%le	double *x	strtod	10
%Le	long double *x	strtod	10
%E	float *x	strtod	10
%lE	double *x	strtod	10
%LE	long double *x	strtod	10
%f	float *x	strtod	10
%lf	double *x	strtod	10
%Lf	long double *x	strtod	10
%g	float *x	strtod	10
%lg	double *x	strtod	10
%Lg	long double *x	strtod	10
%G	float *x	strtod	10
%lG	double *x	strtod	10
%LG	long double *x	strtod	10
%i	int *x	strtol	0
%hi	short *x	strtol	0
%li	long *x	strtol	0
%n	int *x		
%hn	short *x		
%ln	long *x		
%o	unsigned int *x	strtoul	8
%ho	unsigned short *x	strtoul	8
%lo	unsigned long *x	strtoul	8
%p	void **x		
%s	char x[]		
%ls	wchar_t x[]		
%u	unsigned int *x	strtoul	10
%hu	unsigned short *x	strtoul	10
%lu	unsigned long *x	strtoul	10
%x	unsigned int *x	strtoul	16
%hx	unsigned short *x	strtoul	16
%lx	unsigned long *x	strtoul	16
%X	unsigned int *x	strtoul	16
%hX	unsigned short *x	strtoul	16
%lX	unsigned long *x	strtoul	16
%[...]	char x[]		
%l[...]	wchar_t x[]		
%%	**none**		

The scan conversion specifier (or scan set) determines any behavior not summarized in this table. In the following descriptions, examples follow each of the scan conversion specifiers. In each example, the function `sscanf` matches the **bold** characters.

You write **%c** to store the matched input characters in an array object. If you specify no field width *w*, then *w* has the value one. The match does not skip leading white space. Any sequence of *w* characters matches the conversion pattern.

```
sscanf("129E-2", "%c", &c)            stores '1'
sscanf("129E-2", "%2c", &c[0])        stores '1', '2'
```

For a wide stream, conversion occurs as if by repeatedly calling `wcrtomb`, beginning in the initial conversion state.

```
swscanf(L"129E-2", L"%c", &c)         stores '1'
```

You write **%lc** to store the matched input characters in an array object, with elements of type wchar_t. If you specify no field width *w*, then *w* has the value one. The match does not skip leading white space. Any sequence of *w* characters matches the conversion pattern. For a byte stream, conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state.

```
sscanf("129E-2", "%lc", &c)           stores L'1'
sscanf("129E-2", "%2lc", &c)          stores L'1', L'2'
swscanf(L"129E-2", L"%lc", &c)        stores L'1'
```

You write **%d**, **%i**, **%o**, **%u**, **%x**, or **%X** to convert the matched input characters as a signed integer and store the result in an integer object.

```
sscanf("129E-2", "%o%d%x", &i, &j, &k) stores 10, 9, 14
```

You write **%e**, **%E**, **%f**, **%g**, or **%G** to convert the matched input characters as a signed fraction, with an optional exponent, and store the result in a floating-point object.

```
sscanf("129E-2", "%e", &f)            stores 1.29
```

You write **%n** to store the number of characters matched (up to this point in the format) in an integer object. The match does not skip leading white space and does not match any input characters.

```
sscanf("129E-2", "12%n", &i)          stores 2
```

You write **%p** to convert the matched input characters as an external representation of a *pointer to void* and store the result in an object of type *pointer to void.* The input characters must match the form generated by the %p print conversion specification.

```
sscanf("129E-2", "%p", &p)            stores, e.g. 0x129E
```

You write **%s** to store the matched input characters in an array object, followed by a terminating null character. If you do not specify a field width *w,* then *w* has a large value. Any sequence of up to *w* non white-space characters matches the conversion pattern.

```
sscanf("129E-2", "%s", &s[0])         stores "129E-2"
```

For a wide stream, conversion occurs as if by repeatedly calling `wcrtomb` beginning in the initial conversion state.

```
swscanf(L"129E-2", L"%s", &s[0])      stores "129E-2"
```

You write **%ls** to store the matched input characters in an array object, with elements of type wchar_t, followed by a terminating null wide character. If you do not specify a field width *w,* then *w* has a large value. Any sequence of up to *w* non white-space characters matches the conversion pattern. For a byte stream, conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state.

```
sscanf("129E-2", "%ls", &s[0])        stores L"129E-2"
swscanf(L"129E-2", L"%ls", &s[0])     stores L"129E-2"
```

You write **%[** to store the matched input characters in an array object, followed by a terminating null character. If you do not specify a field width *w,* then *w* has a large value. The match does not skip leading white space. A sequence of up to *w* characters matches the conversion pattern in the **scan set** that follows. To complete the scan set, you follow the left bracket ([) in the conversion specification with a sequence of zero or more **match** characters, terminated by a right bracket (]).

If you do not write a caret (^) immediately after the [, then each input character must match *one* of the match characters. Otherwise, each input character must not match *any* of the match characters, which begin with the character following the ^. If you write a] immediately after the [or [^, then the] is the first match character, not the terminating]. If you write a minus (-) as other than the first or last match character, an implementation can give it special meaning. It usually indicates a range of characters, in conjunction with the characters immediately preceding or following, as in 0-9 for all the digits.) You cannot specify a null match character.

```
sscanf("129E-2", "[54321]", &s[0])     stores "12"
```

For a wide stream, conversion occurs as if by repeatedly calling wcrtomb, beginning in the initial conversion state.

```
swscanf(L"129E-2", L"[54321]", &s[0])  stores "12"
```

You write **%l[** to store the matched input characters in an array object, with elements of type wchar_t, followed by a terminating null wide character. If you do not specify a field width *w,* then *w* has a large value. The match does not skip leading white space. A sequence of up to *w* characters matches the conversion pattern in the scan set that follows.

For a byte stream, conversion occurs as if by repeatedly calling mbrtowc, beginning in the initial conversion state.

```
sscanf("129E-2", "l[54321]", &s[0])    stores L"12"
swscanf(L"129E-2", L"l[54321]", &s[0]) stores L"12"
```

You write **%%** to match the percent character (%). The function does not store a value.

```
sscanf("%  0XA", "%% %i")              stores 10
```

Formatted Output

Print Formats Print Functions Print Conversion Specifiers

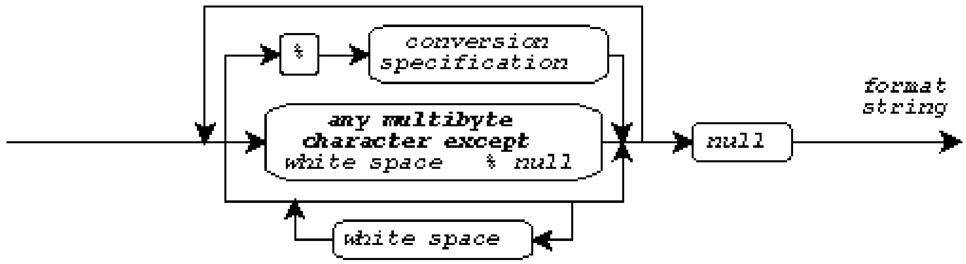
Several library functions help you convert data values from encoded internal representations to text sequences that are generally readable by people. You provide a format string as the value of the format argument to each of these functions, hence the term **formatted output**. The functions fall into two categories.

The **byte print functions** (declared in <stdio.h>) convert internal representations to sequences of type *char,* and help you compose such sequences for display: fprintf, printf, sprintf, vfprintf, vprintf, and vsprintf. For these function, a format string is a multibyte string that begins and ends in the initial shift state.

The **wide print functions** (declared in <wchar.h> and hence added with **Amendment 1**) convert internal representations to sequences of type wchar_t, and help you compose such sequences for display: fwprintf, swprintf, wprintf, vfwprintf, vswprintf, and vwprintf. For these functions, a format string is a wide-character string. In the descriptions that follow, a wide character wc from a format string or a stream is compared to a specific (byte) character c as if by evaluating the expression wctob(wc) == c.

Print Formats

A **format string** has the same syntax for both the print functions and the scan functions, as shown in the diagram.



A format string consists of zero or more **conversion specifications** interspersed with literal text and **white space**. White space is a sequence of one or more characters `c` for which the call `isspace(c)` returns nonzero. (The characters defined as white space can change when you change the LC_CTYPE locale category.) For the print functions, a conversion specification is one of the print conversion specifications described below.

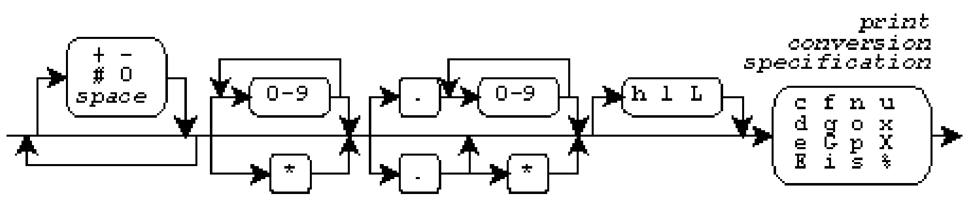
A print function scans the format string once from beginning to end to determine what conversions to perform. Every print function accepts a varying number of arguments, either directly or under control of an argument of type va_list. Some print conversion specifications in the format string use the next argument in the list. A print function uses each successive argument no more than once. Trailing arguments can be left unused.

In the description that follows:

- **integer conversions** are the **conversion specifiers** that end in `d`, `i`, `o`, `u`, `x`, or `X`
- **floating-point conversions** are the conversion specifiers that end in `e`, `E`, `f`, `g`, or `G`

Print Functions

For the print functions, literal text or white space in a format string generates characters that match the characters in the format string. A **print conversion specification** typically generates characters by converting the next argument value to a corresponding text sequence. A print conversion specification has the format:



Following the percent character (%) in the format string, you can write zero or more **format flags**:

- **-** — to left-justify a conversion
- **+** — to generate a plus sign for signed values that are positive
- **space** — to generate a `space` for signed values that have neither a plus nor a minus sign

- **#** — to prefix `0` on an `o` conversion, to prefix `0x` on an `x` conversion, to prefix `0X` on an `X` conversion, or to generate a decimal point and fraction digits that are otherwise suppressed on a floating-point conversion
- **0** — to pad a conversion with leading zeros after any sign or prefix, in the absence of a minus (**-**) format flag or a specified precision

Following any format flags, you can write a **field width** that specifies the minimum number of characters to generate for the conversion. Unless altered by a format flag, the default behavior is to pad a short conversion on the left with `space` characters. If you write an asterisk (*) instead of a decimal number for a field width, then a print function takes the value of the next argument (which must be of type *int*) as the field width. If the argument value is negative, it supplies a **-** format flag and its magnitude is the field width.

Following any field width, you can write a dot (**.**) followed by a **precision** that specifies one of the following: the minimum number of digits to generate on an integer conversion; the number of fraction digits to generate on an `e`, `E`, or `f` conversion; the maximum number of significant digits to generate on a `g` or `G` conversion; or the maximum number of characters to generate from a C string on an `s` conversion.

If you write an `*` instead of a decimal number for a precision, a print function takes the value of the next argument (which must be of type *int*) as the precision. If the argument value is negative, the default precision applies. If you do not write either an `*` or a decimal number following the dot, the precision is zero.

Print Conversion Specifiers

Following any precision, you must write a one-character **print conversion specifier**, possibly preceded by a one-character qualifier. Each combination determines the type required of the next argument (if any) and how the library functions alter the argument value before converting it to a text sequence. The integer and floating-point conversions also determine what base to use for the text representation. If a conversion specifier requires a precision *p* and you do not provide one in the format, then the conversion specifier chooses a default value for the precision. The following table lists all defined combinations and their properties.

Conversion Specifier	Argument Type	Converted Value	Default Base	Pre-cision
%c	int x	(unsigned char)x		
%lc	wint_t x	wchar_t a[2] = {x}		
%d	int x	(int)x	10	1
%hd	int x	(short)x	10	1
%ld	long x	(long)x	10	1
%e	double x	(double)x	10	6
%Le	long double x	(long double)x	10	6
%E	double x	(double)x	10	6
%LE	long double x	(long double)x	10	6
%f	double x	(double)x	10	6
%Lf	long double x	(long double)x	10	6
%g	double x	(double)x	10	6
%Lg	long double x	(long double)x	10	6
%G	double x	(double)x	10	6
%LG	long double x	(long double)x	10	6
%i	int x	(int)x	10	1
%hi	int x	(short)x	10	1
%li	long x	(long)x	10	1
%n	int *x			
%hn	short *x			
%ln	long *x			

%o	int x	(unsigned int)x	8	1
%ho	int x	(unsigned short)x	8	1
%lo	long x	(unsigned long)x	8	1
%p	void *x	(void *)x		
%s	char x[]	x[0]...		**large**
%ls	wchar_t x[]	x[0]...		**large**
%u	int x	(unsigned int)x	10	1
%hu	int x	(unsigned short)x	10	1
%lu	long x	(unsigned long)x	10	1
%x	int x	(unsigned int)x	16	1
%hx	int x	(unsigned short)x	16	1
%lx	long x	(unsigned long)x	16	1
%X	int x	(unsigned int)x	16	1
%hX	int x	(unsigned short)x	16	1
%lX	long x	(unsigned long)x	16	1
%%	**none**	'%'		

The print conversion specifier determines any behavior not summarized in this table. In the following descriptions, *p* is the precision. Examples follow each of the print conversion specifiers. A single conversion can generate up to 509 characters.

You write **%c** to generate a single character from the converted value.

```
printf("%c", 'a')                  generates a
printf("<%3c|%-3c>", 'a', 'b')     generates <  a|b  >
```

For a wide stream, conversion of the character `x` occurs as if by calling `btowc(x)`.

```
wprintf(L"%c", 'a')                generates btowc(a)
```

You write **%lc** to generate a single character from the converted value. Conversion of the character `x` occurs as if it is followed by a null character in an array of two elements of type wchar_t converted by the conversion specification ls.

```
printf("%lc", L'a')                generates a
wprintf(L"lc", L'a')               generates L'a'
```

You write **%d**, **%i**, **%o**, **%u**, **%x**, or **%X** to generate a possibly signed integer representation. `%d` or `%i` specifies signed decimal representation, `%o` unsigned octal, `%u` unsigned decimal, `%x` unsigned hexadecimal using the digits `0-9` and `a-f`, and `%X` unsigned hexadecimal using the digits `0-9` and `A-F`. The conversion generates at least *p* digits to represent the converted value. If *p* is zero, a converted value of zero generates no digits.

```
printf("%d %o %x", 31, 31, 31)     generates 31 37 1f
printf("%hu", 0xffff)              generates 65535
printf("%#X %+d", 31, 31)          generates 0X1F +31
```

You write **%e** or **%E** to generate a signed fractional representation with an exponent. The generated text takes the form ±*d.dddE*±*dd*, where ± is either a plus or minus sign, *d* is a decimal digit, the dot (.) is the decimal point for the current locale, and *E* is either `e` (for `%e` conversion) or `E` (for `%E` conversion). The generated text has one integer digit, a decimal point if *p* is nonzero or if you specify the `#` format flag, *p* fraction digits, and at least two exponent digits. The result is rounded. The value zero has a zero exponent.

```
printf("%e", 31.4)                 generates 3.140000e+01
printf("%.2E", 31.4)               generates 3.14E+01
```

You write **%f** to generate a signed fractional representation with no exponent. The generated text takes the form ±*d.ddd*, where ± is either a plus or minus sign, *d* is a decimal digit, and the dot (.) is the decimal point for the current locale. The generated text has at least one integer digit, a decimal point if *p* is nonzero or if you specify the `#` format flag, and *p* fraction digits. The result is rounded.

```
printf("%f", 31.4)              generates 31.400000
printf("%.0f %#.0f", 31.0, 31.0)generates 31 31.
```

You write **%g** or **%G** to generate a signed fractional representation with or without an exponent, as appropriate. For `%g` conversion, the generated text takes the same form as either `%e` or `%f` conversion. For `%G` conversion, it takes the same form as either `%E` or `%f` conversion. The precision *p* specifies the number of significant digits generated. (If *p* is zero, it is changed to 1.) If `%e` conversion would yield an exponent in the range [-4, *p*), then `%f` conversion occurs instead. The generated text has no trailing zeros in any fraction and has a decimal point only if there are nonzero fraction digits, unless you specify the # format flag.

```
printf("%.6g", 31.4)            generates 31.4
printf("%.1g", 31.4)            generates 3.14e+01
```

You write **%n** to store the number of characters generated (up to this point in the format) in the object of type *int* whose address is the value of the next successive argument.

```
printf("abc%n", &x)                     stores 3
```

You write **%p** to generate an external representation of a *pointer to void.* The conversion is implementation defined.

```
printf("%p", (void *)&x)        generates, e.g. F4C0
```

You write **%s** to generate a sequence of characters from the values stored in the argument C string.

```
printf("%s", "hello")           generates hello
printf("%.2s", "hello")         generates he
```

For a wide stream, conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state. The conversion generates no more than *p* characters, up to but not including the terminating null character.

```
wprintf(L"%s", "hello")         generates hello
```

You write **%ls** to generate a sequence of characters from the values stored in the argument wide-character string. For a byte stream, conversion occurs as if by repeatedly calling `wcrtomb`, beginning in the initial conversion state, so long as complete multibyte characters can be generated. The conversion generates no more than *p* characters, up to but not including the terminating null character.

```
printf("%ls", L"hello")         generates hello
wprintf(L"%.2s", L"hello")      generates he
```

You write **%%** to generate the percent character (`%`).

```
printf("%%")                    generates %
```

Random Number Generators

A **random number generator** is an object that produces a sequence of pseudo-random values. A generator that produces values uniformly distributed within a specified range is an engine. An engine can be combined with a distribution, either by passing the engine as an argument to the distribution's `operator()` or by using a variate_generator, to produce values that are distributed in a manner defined by the distribution.

Most of the random number generators are templates whose parameters customize the generator. The descriptions of generators that take a type as an argument use common template parameter names to describe some of the properties of the type permitted as an actual argument type:

- **IntType** — indicates a signed or unsigned integral type
- **UIntType** — indicates an unsigned integral type
- **RealType** — indicates a floating-point type

An **engine** is a TR1 class or template class whose instances act as a source of random numbers uniformly distributed between a minimum and maximum value. An engine can be a simple engine or a compound engine. Every engine has the following members:

- typedef numeric-type result_type — the type returned by the generator's operator().
- result_type min() — returns the minimum value returned by the generator's operator().
- result_type max() — returns the maximum value returned by the generator's operator(). When result_type is an integral type this is the maximum value that can actually be returned; when result_type is a floating-point type this is the smallest value greater than all values that can be returned.
- void seed() — the **seed function** seeds the engine with default seed values.
- template <class InIt> void seed(InIt& first, InIt last) — the **seed function** seeds the engine with values of type unsigned long from the half-open sequence pointed to by [first, last). If the sequence is not long enough to fully initialize the engine the function stores the value last in first and throws an object of type std::invalid_argument.
- result_type operator()() — returns values uniformly distributed between min() and max().

In addition, every engine has a **state** that determines the sequence of values that will be generated by subsequent calls to operator(). The states of two objects of the same type can be compared with operator== and operator!=; if the two states compare equal the objects will generate the same sequence of values. The state of an object can be saved to a stream as a sequence of 32-bit unsigned values with the object's operator<<; the state is not changed by saving it. A saved state can be read into an object of the same type with operator>>.

A **distribution** is a TR1 class or template class whose instances transform a stream of uniformly distributed random numbers obtained from an engine into a stream of random numbers with a particular distribution. Every distribution has the following members:

- typedef numeric-type input_type — the type that the engine passed to operator() should return.
- typedef numeric-type result_type — the type returned by the distribution's operator().
- void reset() — discards any cached values, so that the result of the next call to operator() will not depend on any values obtained from the engine prior to the call.
- template <class Engine> result_type operator()(Engine& eng) — returns values distributed in accordance with the distribution's definition, using eng as a source of uniformly distributed random values.

A **simple engine** is an engine that produces random numbers directly. This library provides one class whose objects are simple engines. It also provides four class templates which can be instantiated with values that provide parameters for the algorithm they implement, and nine predefined instances of those class templates. Objects of these types are also simple engines.

A **compound engine** is an engine that obtains random numbers from one or more simple engines and generates a stream of uniformly distributed random numbers from those values. The library provides class templates for two compound engines.

The library can be built as a checked version and as an unchecked version. The checked version uses a macro similar to C's `assert` macro to test the conditions marked as ***Precondition***s in the functional descriptions. When a failure is detected the macro writes an error message to standard out and calls the function `std::tr1::_Rng_abort`. For runtime debugging, set a breakpoint on that function.

To use the checked version, define either the macro `_RNG_CHECK` or the macro `_DEBUG` to a non-zero numeric value in all code that uses the library.

Regular Expressions

A **regular expression** is a sequence of characters that can match one or more target sequences of characters, according to a regular expression grammar. This implementation supports the following regular expression grammars:

- **BRE —** Basic Regular Expressions, defined by the **POSIX Standard, Part 1** (ISO/IEC 9945-1:2003)
- **ERE —** Extended Regular Expressions, also defined by the POSIX Standard, Part 1
- **ECMAScript —** ECMAScript regular expressions, as defined by the **ECMAScript Language Specification** (Ecma-262)
- **awk —** regular expressions as used in the *awk* utility, defined by the **POSIX Standard, Part 3** (ISO/IEC 9945-3:2003)
- **grep —** regular expressions as used in the *grep* utility, also defined by the POSIX Standard, Part 3
- **egrep —** regular expressions as used in the *grep* utility with the *-E* option, also defined by the POSIX Standard, Part 3

This document describes each of these grammars as provided in this implementation. Most of the differences between the grammars are in the regular expression features that are supported. When features are not supported by all of the grammars the text describing those features lists the grammars that support them. In some cases the differences between the grammars are in the syntax used to describe a feature (for example, *BRE* and *grep* require a backslash in front of a left parenthesis that marks the beginning of a group and the others do not). In these cases the differences are described as part of the description of the feature.

Regular Expression Grammar

Element

An **element** can be any of the following:

- An ordinary character, which matches the same character in the target sequence
- A wildcard character, '**.**', which matches any character in the target sequence except a newline

- A bracket expression, of the form "[*expr*]", which matches a character or a **collation element** in the target sequence that is also in the set defined by the expression *expr*, or of the form "**[^***expr***]**", which matches a character or a collation element in the target sequence that is not in the set defined by the expression *expr*. The expression *expr* can consist of any combination of any number of each of the following.
  - An **individual character**, which adds that character to the set defined by *expr*.
  - A character range, of the form "*ch1***-***ch2*", which adds all of the characters represented by values in the closed range [*ch1*, *ch2*] to the set defined by *expr*.
  - A character class, of the form "**[:***name***:]**", which adds all of the characters in the named class to the set defined by *expr*.
  - An equivalence class, of the form "**[=***elt***=]**", which adds the collating elements that are equivalent to *elt* to the set defined by *expr*.
  - A collating symbol, of the form "**[.***elt***.]**", which adds the collation element *elt* to the set defined by *expr*.
- An anchor, either '**^**' or '**$**', which matches the beginning or the end of the target sequence, respectively
- A capture group, of the form "**("Subexpression" on page 500)**", or "**\("Subexpression" on page 500 \)**" in *BRE* and *grep*, which matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters
- An identity escape, of the form "***k***", which matches the character *k* in the target sequence

Examples:
- **"a"** matches the target sequence "a" but none of the target sequences "B", "b", or "c".
- **"."** matches all of the target sequences "a", "B", "b", and "c".
- **"[b-z]"** matches the target sequences "b" and "c" but does not match the target sequence "a" or the target sequence "B".
- **"[:lower:]"** matches the target sequences "a", "b", and "c" but does not match the target sequence "B".
- **"(a)"** matches the target sequence "a" and associates capture group 1 with the subsequence "a",but does not match any of the target sequences "B", "b", or "c".

In *ECMAScript*, *BRE*, and *grep* an element can also be:
- a back reference, of the form "***dd***" where *dd* represents a decimal value N, which matches a sequence of characters in the target sequence that is the same as the sequence of characters matched by the Nth capture group.

For example:
- **"(a)\1"** matches the target sequence "aa" because the first (and only) capture group matches the initial sequence "a" and the \1 then matches the final sequence "a".

In *ECMAScript*, an element can also be any of the following:
- A non-capture group, of the form "**(?: "Subexpression" on page 500)**", which matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters
- a limited file format escape, of the form "**\f**", "**\n**", "**\r**", "**\t**", or "**\v**"; these match a form feed, newline, carriage return, horizontal tab, and vertical tab, respectively, in the target sequence.

- A positive assert, of the form "**(?= "Subexpression" on page 500)**", which matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters, but does not change the match position in the target sequence.
- A negative assert, of the form "**(?! "Subexpression" on page 500)**", which matches any sequence of characters in the target sequence that does not match the pattern between the delimiters, and does not change the match position in the target sequence.
- A hexadecimal escape sequence, of the form "**\x*hh***", which matches a character in the target sequence whose representation is the value represented by the two hexadecimal digits *hh*.
- A unicode escape sequence, of the form "**\u*hhhh***", which matches a character in the target sequence whose representation is the value represented by the four hexadecimal digits *hhhh*.
- A control escape sequence, of the form "**\c*k***", which matches the control character named by the character *k*.
- A word boundary assert, of the form "**\b**", which matches if the current position in the target sequence is immediately after a word boundary.
- A negative word boundary assert, of the form "**\B**", which matches if the current position in the target sequence is not immediately after a word boundary.
- A dsw character escape, of the form "**\d**", "**\D**", "**\s**", "**\S**", "**\w**", "**\W**", which provides a short name for a character class.

For example:
- **"(?:a)"** matches the target sequence "a", but "(?:a)\1" is invalid, because there is no capture group 1.
- **"(?=a)a"** matches the target sequence "a". The positive assert matches the initial sequence "a" in the target sequence and the final "a" in the regular expression matches the initial sequence "a" in the target sequence.
- **"(?!a)a"** does not match the target sequence "a".
- **"a\b."** matches the target sequence "a~" but does not match the target sequence "ab".
- **"a\B."** matches the target sequence "ab" but does not match the target sequence "a~".

In *awk*, an element can also be one of the following:
- A file format escape, of the form "****", "**\a**", "**\b**", "**\f**", "**\n**", "**\r**", "**\t**", or "**\v**"; these match a backslash, alert, backspace, form feed, newline, carriage return, horizontal tab, and vertical tab, respectively, in the target sequence.
- An octal escape sequence, of the form "*ooo*", which matches a character in the target sequence whose representation is the value represented by the one, two, or three octal digits *ooo*.

Repetition

Any element other than a positive assert, a negative assert, or an anchor can be followed by a **repetition count**. The most general form of repetition count takes the form "**{*min,max*}**", or "**\{*min,max*\}**" in *BRE* and *grep*. An element followed by this form of repetition count matches at least *min* and no more than *max* successive occurrences of a sequence that matches the element.

For example:

- **“a{2,3}”** matches the target sequence “aa” and the target sequence “aaa”, but not the target sequence “a” or the target sequence “aaaa”.

A repetition count can also take one of the following forms:
- “**{*min*}**”, or “**\{*min*\}**” in *BRE* and *grep*, which is equivalent to “{*min*,*min*}”.
- “**{*min*,}**”, or “**\{*min*,\}**” in *BRE* and *grep*, which is equivalent to “{*min*,unbounded}”.
- “*****”, which is equivalent to “{0,unbounded}”.

Examples:
- **“a{2}”** matches the target sequence “aa” but not the target sequence “a” or the target sequence “aaa”.
- **“a{2,}”** matches the target sequence “aa”, the target sequence “aaa”, and so on, but does not match the target sequence “a”.
- **“a*”** matches the target sequence “”, the target sequence “a”, the target sequence “aa”, and so on.

For all grammars except *BRE* and *grep*, a repetition count can also take one of the following forms:
- “**?**”, which is equivalent to “{0,1}”.
- “**+**”, which is equivalent to “{1,unbounded}”.

Examples:
- **“a?”** matches the target sequence “” and the target sequence “a”, but not the target sequence “aa”.
- **“a+”** matches the target sequence “a”, the target sequence “aa”, and so on, but not the target sequence “”.

Finally, in *ECMAScript*, all of the preceding forms of repetition count can be followed by the character '**?**', which designates a non-greedy repetition.

Concatenation

Regular expression elements, with our without repetition counts, can be concatenated to form longer regular expressions. Such an expression matches a target sequence that is a concatenation of sequences matched by the individual elements.

For example:
- **“a{2,3}b”** matches the target sequence “aab" and the target sequence ”aaab“, but does not match the target sequence ”ab" or the target sequence “aaaab”.

Alternation

For all regular expression grammars except *BRE* and *grep*, a concatenated regular expression can be followed by the character '|' and another concatenated regular expression, which can be followed by another '|' and another concatenated regular expression, and so on. Such an expression matches any target sequence that matches one or more of the concatenated regular expressions. When more than one of the concatenated regular expressions matches the target sequence, *ECMAScript* chooses the first of the concatenated regular expressions that matches the sequence as the match (first match); the other regular expression grammars choose the one that results in the longest match.

For example:

- **“ab|cd”** matches the target sequence “ab” and the target sequence “cd”, but does not match the target sequence “abd” or the target sequence “acd”.

In *grep* and *egrep*, a newline character ('**\n**') can be used to separate alternations.

Subexpression

A **subexpression** is a **concatenation** in *BRE* and *grep*, or an **alternation** in the other regular expression grammars.

Grammar Summary

Element	BRE	ERE	ECMA	grep	egrep	awk
alternation using '|'		+	+		+	+
alternation using '\n'				+	+	
anchor	+	+	+	+	+	+
back reference	+		+	+		
bracket expression	+	+	+	+	+	+
capture group using “()”		+	+		+	+
capture group using “\(\)”	+			+		
control escape sequence			+			
dsw character escape			+			
file format escape			+			+
hexadecimal escape sequence			+			
identity escape	+	+	+	+	+	+
negative assert			+			
negative word boundary assert			+			
non-capture group			+			
non-greedy repetition			+			
octal escape sequence						+

Element	BRE	ERE	ECMA	grep	egrep	awk
ordinary character	+	+	+	+	+	+
positive assert			+			
repetition using "{}"		+	+		+	+
repetition using "\{\}"	+			+		
repetition using '*'	+	+	+	+	+	+
repetition using '?' and '+'		+	+		+	+
unicode escape sequence			+			
wildcard character	+	+	+	+	+	+
word boundary assert			+			

Semantic Details

Anchor

An anchor matches a position in the target string and not a character. A **'^'** matches the beginning of the target string, and a **'$'** matches the end of the target string.

Back Reference

A **back reference** is a backslash followed by a decimal value N. It matches the contents of the Nth capture group. The value of `N` must not be greater than the number of capture groups that precede the back reference. In *BRE* amd *grep* the value of N is determined by the decimal digit that follows the backslash. In *ECMAScript* the value of N is determined by all of the decimal digits that immediately follow the backslash. Thus, in *BRE* and *grep* the value of N is never greater than 9, even if the regular expression has more than nine capture groups. In *ECMAScript* the value of N is unbounded.

Examples:

- **"((a+)(b+))(c+)\3"** matches the target sequence "aabbbcbbb". The back reference "\3" matches the text in the third capture group, that is, the "(b+)". It does not match the target sequence "aabbbcbb".
- **"(a)\2"** is not valid.
- **"(b(((((((((a))))))))))\10"** has a different meaning in *BRE* and in *ECMAScript*. In *BRE* the back reference is "\1". It matches the contents of the first capture group (i.e. the one beginning with "(b" and ending with the final ")" preceding the back reference), and the final '0' matches the ordinary character '0'. In *ECMAScript* the back reference is "\10". It matches the tenth capture group (i.e. the innermost one).

Bracket Expression

A **bracket expression** defines a set of characters and collating elements. If the bracket expression begins with the character **'^'** the match succeeds if none of the elements in the set matches the current character in the target sequence. Otherwise, the match succeeds if any of the elements in the set matches the current character in the target sequence.

The set of characters can be defined by listing any combination of individual characters, character ranges, character classes, equivalence classes, and collating symbols.

Capture Group

A **capture group** marks its contents as a single unit in the regular expression grammar and labels the target text that matches its contents. The label associated with each capture group is a number, determined by counting the left parentheses marking capture groups up to and including the left parenthesis marking the current capture group. In this implementation, the maximum number of capture groups is 31.

Examples:
- **"ab+"** matches the target sequence "abb" but not the target sequence "abab".
- **"(ab)+"** does not match the target sequence "abb" but matches the target sequence "abab".
- **"((a+)(b+))(c+)"** matches the target sequence "aabbbc" and associates capture group 1 with the subsequence "aabbb", capture group 2 with the subsequence "aa", capture group 3 with "bbb", and capture group 4 with the subsequence "c".

Character Class

A **character class** in a bracket expression adds all the characters in the named class to the character set defined by the bracket expression. To create a character class, use "[:" followed by the name of the class followed by ":]". Internally, names of character classes are recognized by calling `id = traits.lookup_classname`. A character `ch` belongs to such a class if `traits.isctype(ch, id)` returns true. The default `regex_traits` template supports the following class names:
- **"alnum"** — lowercase letters, uppercase letters, and digits;
- **"alpha"** — lowercase letters and uppercase letters;
- **"blank"** — space or tab;
- **"cntrl"** — the file format escape characters;
- **"digit"** — digits;
- **"graph"** — lowercase letters, uppercase letters, digits, and punctuation;
- **"lower"** — lowercase letters;
- **"print"** — lowercase letters, uppercase letters, digits, punctuation, and space;
- **"punct"** — punctuation;
- **"space"** — space;
- **"upper"** — uppercase characters;
- **"xdigit"** — digits, 'a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F';
- **"d"** — same as digit;
- **"s"** — same as space;
- **"w"** — same as alnum.

Character Range

A **character range** in a bracket expression adds all the characters in the range to the character set defined by the bracket expression. To create a character range put the character '-' between the first and last characters in the range. This puts all the characters whose numeric value is greater than or equal to the numeric value of the first character and less than or equal to the numeric value of the last character into the set. Note that this set of added characters depends on the platform-specific representation of characters. If the character '-' occurs at the beginning or end of a bracket expression or as the first or last character of a character range it represents itself.

Examples:

- **“[0-7]”** represents the set of characters { '0', '1', '2', '3', '4', '5', '6', '7' }. It matches the target sequences “0”, “1”, etc., but not “a”.
- **“[h-k]”** represents the set of characters { 'h', 'i', 'j', 'k' } on systems that use the ASCII character encoding; it matches the target sequences “h", ”i“, etc., but not ”\x8A“ or ”0“.
- **”[h-k]“** represents the set of characters { 'h', 'i', '\x8A', '\x8B', '\x8C', '\x8D', '\x8E', '\x8F', '\x90', 'j', 'k' } on systems that use the EBCDIC character encoding ('h' is encoded as 0x88 and 'k' is encoded as 0x92). It matches the target sequences ”h", “i”, “\x8A”, etc., but not “0”.
- **“[-0-24]”** represents the set of characters { '-', '0', '1', '2', '4' }.
- **“[0-2-]”** represents the set of characters { '0', '1', '2', '-' }.
- **“[+—]”** on systems that use ASCII represents the set of characters { '+', ',', '-' }.

When using **locale-sensitive ranges**, however, the characters in a range are determined by the collation rules for the locale. Characters that collate after the first character in the definition of the range and before the last character in the definition of the range are in the set, as are the two end characters.

Collating Element

A **collating element** is a multi-character sequence that is treated as a single character.

Collating Symbol

A **collating symbol** in a bracket expression adds a collating element to the set defined by the bracket expression. To create a collating symbol, use “[.” followed by the collating element followed by “.]”.

Control Escape Sequence

A **control escape sequence** is a backslash followed by the letter 'c' followed by one of the letters 'a' through 'z' or 'A' through 'Z'. It matches the ASCII control character named by that letter.

For example,

- **“\ci”** matches the target sequence “\x09”, because <ctrl-i> has the value 0x09.

DSW Character Escape

A **dsw character escape** is a short name for a character class.

Escape Sequence	Equivalent Named Class	Default Named Class
“**\d**”	“[[:d:]]”	“[[:digit:]]”
“**\D**”	“[^[:d:]]”	“[^[:digit:]]”

Escape Sequence	Equivalent Named Class	Default Named Class
"**\s**"	"[[:s:]]"	"[[:space:]]"
"**\S**"	"[^[:s:]]"	"[^[:space:]]"
"**\w**"	"[[:w:]]"	"[a-zA-Z0-9_]"*
"**\W**"	"[^[:w:]]"	"[^a-zA-Z0-9_]"*
*ASCII character set		

Equivalence Class

An **equivalence class** in a bracket expression adds all the characters and collating elements that are equivalent to the collating element in the equivalence class definition to the set defined by the bracket expression. To create an equivalence class, use "[=" followed by a collating element followed by "=]". Internally, two collating elements `elt1` and `elt2` are equivalent if `traits.transform_primary(elt1.begin(), elt1.end()) == traits.transform_primary(elt2.begin(), elt2.end())`.

File Format Escape

A **file format escape** consists of the usual C language character escape sequences, "\\", "\a", "\b", "\f", "\n", "\r", "\t", "\v", with their usual meanings, namely, backslash, alert, backspace, form feed, newline, carriage return, horizontal tab, and vertical tab, respectively. In *ECMAScript* "\a" and "\b" are not allowed. ("\\" is allowed, but technically it's an identity escape, not a file format escape).

Hexadecimal Escape Sequence

A **hexadecimal escape sequence** is a backslash followed by the letter 'x' followed by two hexadecimal digits (0-9a-fA-F). It matches a character in the target sequence with the value specified by the two digits.

For example,

- **"\x41"** matches the target sequence "A" when the ASCII character encoding is used.

Identity Escape

An **identity escape** is a backslash followed by a single character. It matches that character. It is needed when the character has a special meaning; using the identity escape removes the special meaning.

For example,

- **"a*"** matches the target sequence "aaa" but does not match the target sequence "a*"
- **"a*"** does not match the target sequence "aaa" but does match the target sequence "a*"

The set of characters allowed in an identity escape depends on the regular expression grammar.

- *BRE*, *grep* — { '(', ')', '{', '}', '.', '[', '\', '*', '^', '$' }.
- *ERE*, *egrep* — { '(', ')', '{', '.', '[', '\', '*', '^', '$', '+', '?', '|' }.
- *awk* — *ERE* plus { '"', '/' }.
- *ECMAScript* — all characters except those that can be part of an identifier. Roughly speaking, this is letters, digits, '$', '_', and unicode escape sequences. For full details see the ECMAScript Language Specification.

Individual Character

An **individual character** in a bracket expression adds that character to the character set defined by the bracket expression. A '^' anywhere other than at the beginning of a bracket expression represents itself.

Examples:

- **"[abc]"** matches the target sequences "a", "b", and "c" but not the sequence "d".
- **"[^abc]"** matches the target sequence "d", but not "a", "b", or "c".
- **"[a^bc]"** matches the target sequences "a", "b", "c", and "^" but not the sequence "d".

In all the regular expression grammars except *ECMAScript* if a ']' is the first character following the opening '[' or the first character following an initial '^' it represents itself.

Examples:

- **"[]a"** is invalid, because there is no ']' to end the bracket expression.
- **"[]abc]"** matches the target sequences "a", "b", "c", and "]" but not the sequence "d".
- **"[^]abc]"** matches the target sequence "d", but not "a", "b", "c", or "]".

In *ECMAScript* use '\]' to represent the character ']' in a bracket expression.

Examples:

- **"[]a"** matches the target sequence "a" because the bracket expression is empty.
- **"[\]abc]"** matches the target sequences "a", "b", "c", and "]" but not the sequence "d".

Negative Assert

A **negative assert** matches anything but its contents; it does not consume any characters in the target sequence.

For example,

- **"(?!aa)(a*)"** matches the target sequence "a" and associates capture group 1 with the subsequence "a". It does not match the target sequence "aa" or the target sequence "aaa".

Negative Word Boundary Assert

A **negative word boundary assert** matches if the current position in the target string is not immediately after a word boundary.

Non-capture Group

A **non-capture group** marks its contents as a single unit in the regular expression grammar, but does not label the target text.

For example,

- **"(a)(?:b)*(c)** matches the target text "abbc" and associates capture group 1 with the subsequence "a" and capture group 2 with the subsequence "c".

Non-greedy Repetition

A **non-greedy** repetition consumes the shortest subsequence of the target sequence that matches the pattern. A **greedy** repetition consumes the longest.

For example,

- **“(a+)(a*b)”** matches the target sequence “aaab”. When using a non-greedy repetition it associates capture group 1 with the subsequence “a” at the beginning of the target sequence and capture group 2 with the subsequence “aab” at the end of the target sequence. When using a greedy match it associates capture group 1 with the subsequence “aaa” and capture group 2 with the subsequence “b”.

Octal Escape Sequence

An **octal escape sequence** is a backslash followed by one, two, or three octal digits (0-7). It matches a character in the target sequence with the value specified by those digits. If all the digits are '0' the sequence is invalid.

For example,

- **“\101”** matches the target sequence “A” when the ASCII character encoding is used.

Ordinary Character

An **ordinary character** is any valid character that doesn't have a special meaning in the current grammar.

In *ECMAScript* the characters that have special meanings are:

```
^ $ \ . * + ? ( ) [ ] { } |
```

In *BRE* and *grep* the characters that have special meanings are:

```
. [ \
```

In addition, the following characters have special meanings when used in a particular context:

- **'*'** has a special meaning in all cases except when it is the first character in a regular expression or the first character following an initial '^' in a regular expression and when it is the first character of a capture group or the first character following an initial '^' in a capture group.
- **'^'** has a special meaning when it is the first character of a regular expression.
- **'$'** has a special meaning when it is the last character of a regular expression.

In *ERE*, *egrep*, and *awk* the following characters have special meanings:

```
. [ \ ( * + ? { |
```

In addition, the following characters have special meanings when used in a particular context.

- **')'** has a special meaning when it matches a preceding '('.
- **'^'** has a special meaning when it is the first character of a regular expression.
- **'$'** has a special meaning when it is the last character of a regular expression.

An ordinary character matches the same character in the target sequence. By default this means that the match succeeds if the two characters are represented by the same value. In a **case-insensitive** match two characters `ch0` and `ch1` match if `traits.translate_nocase(ch0) == traits.translate_nocase(ch1)`. In a **locale-sensitive** match two characters `ch0` and `ch1` match if `traits.translate(ch0) == traits.translate(ch1)`.

Positive Assert

A **positive assert** matches its contents, but does not consume any characters in the target sequence.

Examples:

- **“(?=aa)(a*)”** matches the target sequence “aaaa” and associates capture group 1 with the subsequence “aaaa”.
- In contrast, **“(aa)(a*)”** matches the target sequence “aaaa” and associates capture group 1 with the subsequence “aa” at the beginning of the target sequence and capture group 2 with the subsequence “aa” at the end of the target sequence.
- **“(?=aa)(a)|(a)”** matches the target sequence “a” and associates capture group 1 with an empty sequence (because the positive assert failed) and capture group 2 with the subsequence “a”. It also matches the target sequence “aa" and associates capture group 1 with the subsequence ”aa“ and capture group 2 with an empty sequence.

Unicode Escape Sequence

A **unicode escape sequence** is a backslash followed by the letter 'u' followed by four hexadecimal digits (0-9a-fA-F). It matches a character in the target sequence with the value specified by the four digits.

For example,

- **”\u0041“** matches the target sequence ”A“ when the ASCII character encoding is used.

Wildcard Character

A **wildcard character** matches any character in the target expression except a newline.

Word Boundary

A **word boundary** occurs in the following situations:

- the current character is at the beginning of the target sequence and the current character is one of the **word characters** *A-Za-z0-9_*
- the current character position is past the end of the target sequence and the last character in the target sequence is one of the **word characters**
- the current character is one of the **word characters** and the preceding character is not
- the current character is not one of the **word characters** and the preceding character is.

Word Boundary Assert

A **word boundary assert** matches if the current position in the target string is immediately after a word boundary.

Matching and Searching

For a regular expresion to **match** a target sequence, the entire regular expression must match the entire target sequence.

For example:

- the regular expression **”bcd“** matches the target sequence ”bcd“ but does not match the target sequence ”abcd“ nor the target sequence ”bcde“.

For a regular expression **search** to succeed there must be a subsequence somewhere in the target sequence that matches the regular expression. The search ordinarily finds the leftmost matching subsequence.

Examples:

- A search for the regular expression **"bcd"** in the target sequence "bcd" succeeds and matches the entire sequence; the same search in the target sequence "abcd" also succeeds and matches the last three characters; the same search in the target sequence "bcde" also succeeds, and matches the first three characters.
- A search for the regular expression **"bcd"** in the target sequence "bcdbcd" succeeds and matches the first three characters.

If there is more than one subsequence that matches at some position in the target sequence there are two ways to choose the matching pattern. **First match** chooses the subsequence that was found first when matching the regular expression. **Longest match** chooses the longest subsequence from the ones that match at that point. If there is more than one subsequence with the maximal length, longest match chooses the subsequence that was found first.

For example:
- a search for the regular expression **"b|bc"** in the target sequence "abcd" matches the subsequence "b" with first match, because the left-hand term of the alternation matched that subsequence and there was no need to try the right-hand term of the alternation; the same search matches "bc" with longest match, because "bc" is longer than "b".

A **partial match** succeeds if the match reaches the end of the target sequence without failing, even if it has not reached the end of the regular expression. Thus, after a partial match succeeds, appending characters to the target sequence could cause a subsequent partial match to fail. After a partial match fails, appending characters to the target sequence cannot cause a subsequent partial match to succeed.

For example, with a partial match:
- **"ab"** matches the target sequence "a" but not "ac".

Format Flags

ECMAScript format rules	sed format rules	Replacement text
"$&"	**"&"**	The character sequence that matched the entire regular expression (`[match[0].first, match[0].second)`)
"$$"		"$"
	"\&"	"&"
"$`" (dollar sign followed by back quote)		The character sequence that precedes the subsequence that matched the regular expression (`[match.prefix().first, match.prefix().second)`)
"$'" (dollar sign followed by forward quote)		The character sequence that follows the subsequence that matched the regular expression (`[match.suffix().first, match.suffix().second)`)

ECMAScript format rules	sed format rules	Replacement text
″**$n**“	″**\n**“	The character sequence that matched the n^{th} (0 <= n <= 9) capture group ([match[n].first, match[n].second)
	″**\\n**“	″\n“
″**$nn**“		The character sequence that matched the nn^{th} (10 <= nn <= 99) capture group ([match[nn].first, match[nn].second)

STL Conventions

The Standard Template Library, or STL, establishes uniform standards for the application of iterators to STL containers or other sequences that you define, by STL algorithms or other functions that you define. This document summarizes many of the conventions used widely throughout the Standard Template Library.

Iterator Conventions

The STL facilities make widespread use of **iterators**, to mediate between the various algorithms and the sequences upon which they act. For brevity in the remainder of this document, the name of an iterator type (or its prefix) indicates the category of iterators required for that type. In order of increasing power, the categories are summarized here as:

- **OutIt —** An **output iterator** X can only have a value V stored indirect on it, after which it *must* be incremented before the next store, as in (*X++ = V), (*X = V, ++X), or (*X = V, X++).
- **InIt —** An **input iterator** X can represent a singular value that indicates end-of-sequence. If an input iterator does not compare equal to its end-of-sequence value, it can have a value V accessed indirect on it any number of times, as in (V = *X). To progress to the next value, or end-of-sequence, you increment it, as in ++X, X++, or (V = *X++). Once you increment *any* copy of an input iterator, none of the other copies can safely be compared, dereferenced, or incremented thereafter.
- **FwdIt —** A **forward iterator** X can take the place of an output iterator (for writing) or an input iterator (for reading). You can, however, read (via V = *X) what you just wrote (via *X = V) through a forward iterator. And you can make multiple copies of a forward iterator, each of which can be dereferenced and incremented independently.
- **BidIt —** A **bidirectional iterator** X can take the place of a forward iterator. You can, however, also decrement a bidirectional iterator, as in –X, X–, or (V = *X–).
- **RanIt —** A **random-access iterator** X can take the place of a bidirectional iterator. You can also perform much the same integer arithmetic on a random-access iterator that you can on an object pointer. For N an integer object, you can write x[N], x + N, x - N, and N + X.

Note that an object pointer can take the place of a random-access iterator, or any other for that matter. All iterators can be assigned or copied. They are assumed to be lightweight objects and hence are often passed and returned by value, not by reference. Note also that none of the operations described above can throw an exception, at least when performed on a valid iterator.

The hierarchy of iterator categories can be summarize by showing three sequences. For write-only access to a sequence, you can use any of:

```
output iterator
    -> forward iterator
    -> bidirectional iterator
    -> random-access iterator
```

The right arrow means ``can be replaced by.'' So any algorithm that calls for an output iterator should work nicely with a forward iterator, for example, but *not* the other way around.

For read-only access to a sequence, you can use any of:

```
input iterator
    -> forward iterator
    -> bidirectional iterator
    -> random-access iterator
```

An input iterator is the weakest of all categories, in this case.

Finally, for read/write access to a sequence, you can use any of:

```
forward iterator
    -> bidirectional iterator
    -> random-access iterator
```

Remember that an object pointer can always serve as a random-access iterator. Hence, it can serve as any category of iterator, so long as it supports the proper read/write access to the sequence it designates.

An iterator `It` other than an object pointer must also define the member types required by the specialization `iterator_traits<It>`. Note that these requirements can be met by deriving `It` from the public base class `iterator`.

This ``algebra'' of iterators is fundamental to practically everything else in the Standard Template Library. It is important to understand the promises, and limitations, of each iterator category to see how iterators are used by containers and algorithms in STL.

Algorithm Conventions

The descriptions of the algorithm template functions employ several shorthand phrases:

- The phrase ``**in the range [A, B)**'' means the sequence of zero or more discrete values beginning with `A` up to but not including `B`. A range is valid only if `B` is **reachable** from `A` — you can store `A` in an object `N` (`N = A`), increment the object zero or more times (`++N`), and have the object compare equal to `B` after a finite number of increments (`N == B`).
- The phrase ``**each N in the range [A, B)**'' means that `N` begins with the value `A` and is incremented zero or more times until it equals the value `B`. The case `N == B` is *not* in the range.
- The phrase ``**the lowest value of N in the range [A, B) such that X**'' means that the condition X is determined for each `N` in the range `[A, B)` until the condition X is met.
- The phrase ``**the highest value of N in the range [A, B) such that X**'' usually means that X is determined for each `N` in the range `[A, B)`. The function stores in `K` a copy of `N` each time the condition X is met. If any such store occurs, the function replaces the final value of `N` (which equals `B`) with the value of `K`. For a

bidirectional or random-access iterator, however, it can also mean that `N` begins with the highest value in the range and is decremented over the range until the condition X is met.

- Expressions such as **X - Y**, where `X` and `Y` can be iterators other than random-access iterators, are intended in the mathematical sense. The function does not necessarily evaluate `operator-` if it must determine such a value. The same is also true for expressions such as **X + N** and **X - N**, where `N` is an integer type.

Several algorithms make use of a predicate, using `operator==`, that must impose an **equivalence relationship** on pairs of elements from a sequence. For all elements `X`, `Y`, and `Z`:

- `X == X` is true.
- If `X == Y` is true, then `Y == X` is true.
- If `X == Y && Y == Z` is true, then `X == Z` is true.

Several algorithms make use of a predicate that must impose a **strict weak ordering** on pairs of elements from a sequence. For the predicate `pr(X, Y)`:

- ``strict'' means that `pr(X, X)` is false
- ``weak'' means that `X` and `Y` have an **equivalent ordering** if `!pr(X, Y) && !pr(Y, X)` (`X == Y` need not be defined)
- ``ordering'' means that `pr(X, Y) && pr(Y, Z)` implies `pr(X, Z)`

Some of these algorithms implicitly use the predicate `X < Y`. Other predicates that typically satisfy the ``strict weak ordering'' requirement are `X > Y`, `less(X, Y)`, and `greater(X, Y)`. Note, however, that predicates such as `X <= Y` and `X >= Y` do *not* satisfy this requirement.

A sequence of elements designated by iterators in the range `[first, last)` is ``**a sequence ordered by `operator<`**'' if, for each `N` in the range `[0, last - first)` and for each `M` in the range `(N, last - first)` the predicate `!(*(first + M) < *(first + N))` is true. (Note that the elements are sorted in *ascending* order.) The predicate function `operator<`, or any replacement for it, must not alter either of its operands. Moreover, it must impose a strict weak ordering on the operands it compares.

A sequence of elements designated by iterators in the range `[first, last)` is ``**a heap ordered by `operator<`**'' if, for each `N` in the range `[1, last - first)` the predicate `!(*first < *(first + N))` is true. (The first element is the largest.) Its internal structure is otherwise known only to the template functions make_heap, pop_heap, and push_heap. As with an ordered sequence, the predicate function `operator<`, or any replacement for it, must not alter either of its operands, and it must impose a strict weak ordering on the operands it compares.

Containers overview

A **container** is an STL template class that manages a sequence of elements. Such elements can be of any object type that supplies a copy constructor, a destructor, and an assignment operator (all with sensible behavior, of course). The destructor may not throw an exception. This document describes the properties required of all such containers, in terms of a generic template class `Cont`. An actual container template class may have additional template parameters. It will certainly have additional member functions.

The STL template container classes are:

- "array" on page 35 (Added with TR1)
- "deque" on page 66
- "list" on page 159
- "map" on page 216
- "multimap" on page 223
- "multiset" on page 330
- "set" on page 336
- "unordered_map" on page 414 (Added with TR1)
- "unordered_multimap" on page 421 (Added with TR1)
- "unordered_set" on page 437 (Added with TR1)
- "unordered_multiset" on page 429 (Added with TR1)
- "vector" on page 470

The Standard C++ Library template class `basic_string` also meets the requirements for a template container class.

Containers

Synopsis

```
namespace std {
template<class T>
    class Cont;

        // TEMPLATE FUNCTIONS
template<class T>
    bool operator==(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    bool operator!=(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    bool operator<(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    bool operator>(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    bool operator<=(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    bool operator>=(
        const Cont<T>& lhs,
        const Cont<T>& rhs);
template<class T>
    void swap(
        Cont<T>& lhs,
        Cont<T>& rhs);
    };
```

Classes

Cont:

Description: The template class describes an object that controls a varying-length sequence of elements, typically of type `T`. The sequence is stored in different ways, depending on the actual container.

A container constructor or member function may call the constructor `T(const T&)` or the function `T::operator=(const T&)`. If such a call throws an exception, the container object is obliged to maintain its integrity, and to rethrow any exception it catches. You can safely swap, assign to, erase, or destroy a container object after it throws one of these exceptions. In general, however, you cannot otherwise predict the state of the sequence controlled by the container object.

A few additional caveats:
- If the expression `~T()` throws an exception, the resulting state of the container object is undefined.
- If the container stores an allocator object `al`, and `al` throws an exception other than as a result of a call to `al.allocate`, the resulting state of the container object is undefined.
- If the container stores a function object `comp`, to determine how to order the controlled sequence, and `comp` throws an exception of any kind, the resulting state of the container object is undefined.

The container classes defined by STL satisfy several additional requirements, as described in the following paragraphs.

Container template class `list` provides deterministic, and useful, behavior even in the presence of the exceptions described above. For example, if an exception is thrown during the insertion of one or more elements, the container is left unaltered and the exception is rethrown.

For *all* the container classes defined by STL, if an exception is thrown during calls to the following member functions:

```
insert // single element inserted
push_back
push_front
```

the container is left unaltered and the exception is rethrown.

For *all* the container classes defined by STL, no exception is thrown during calls to the following member functions:

```
erase // single element erased
pop_back
pop_front
```

Moreover, no exception is thrown while copying an iterator returned by a member function.

The member function `swap` makes additional promises for *all* container classes defined by STL:
- The member function throws an exception only if the container stores an allocator object `al`, and `al` throws an exception when copied.
- References, pointers, and iterators that designate elements of the controlled sequences being swapped remain valid.

An object of a container class defined by STL allocates and frees storage for the sequence it controls through a stored object of type `A`, which is typically a template parameter. Such an allocator object must have the same external interface as an object of class allocator. In particular, A must be the same type as A::rebind<value_type>::other

For *all* container classes defined by STL, the member function:

```
A get_allocator() const;
```

returns a copy of the stored allocator object. Note that the stored allocator object is *not* copied when the container object is assigned. All constructors initialize the value stored in allocator, to A() if the constructor contains no allocator parameter.

According to the C++ Standard a container class defined by STL can assume that:

- All objects of class A compare equal.
- Type A::const_pointer is the same as const T *.
- Type A::const_reference is the same as const T&.
- Type A::pointer is the same as T *.
- Type A::reference is the same as T&.

In this implementation, however, containers do *not* make such simplifying assumptions. Thus, they work properly with allocator objects that are more ambitious:

- All objects of class A need not compare equal. (You can maintain multiple pools of storage.)
- Type A::const_pointer need not be the same as const T *. (A pointer can be a class.)
- Type A::pointer need not be the same as T *. (A const pointer can be a class.)

Synopsis:

```
template<class T<T> >
    class Cont {
public:
    typedef T0 size_type;
    typedef T1 difference_type;
    typedef T2 reference;
    typedef T3 const_reference;
    typedef T4 value_type;
    typedef T5 iterator;
    typedef T6 const_iterator;
    typedef T7 reverse_iterator;
    typedef T8 const_reverse_iterator;
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    void clear();
    void swap(Cont& x);
    };
```

Types:

Cont::const_iterator:

```
typedef T6 const_iterator;
```

The type describes an object that can serve as a constant iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T6`.

Cont::const_reference:

```
typedef T3 const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence. It is described here as a synonym for the unspecified type `T3` (typically `A::const_reference`).

Cont::const_reverse_iterator:

```
typedef T8 const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T8` (typically `reverse_iterator <const_iterator>`).

Cont::difference_type:

```
typedef T1 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the unspecified type `T1` (typically `A::difference_type`).

Cont::iterator:

```
typedef T5 iterator;
```

The type describes an object that can serve as an iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T5`. An object of type `iterator` can be cast to an object of type const_iterator.

Cont::reference:

```
typedef T2 reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence. It is described here as a synonym for the unspecified type `T2` (typically `A::reference`). An object of type `reference` can be cast to an object of type const_reference.

Cont::reverse_iterator:

```
typedef T7 reverse_iterator;
```

The type describes an object that can serve as a reverse iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T7` (typically `reverse_iterator <iterator>`).

Cont::size_type:

```
typedef T0 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the unspecified type `T0` (typically `A::size_type`).

Cont::value_type:

```
typedef T4 value_type;
```

The type is a synonym for the template parameter `T`. It is described here as a synonym for the unspecified type `T4` (typically `A::value_type`).

Member functions:

Cont::begin:

```
const_iterator begin() const;
iterator begin();
```

The member function returns an iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

Cont::clear:

```
void clear();
```

The member function calls `erase( begin(), end())`.

Cont::empty:

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

Cont::end:

```
const_iterator end() const;
iterator end();
```

The member function returns an iterator that points just beyond the end of the sequence.

Cont::erase:

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements of the controlled sequence in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The member functions never throw an exception.

Cont::max_size:

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control, in constant time regardless of the length of the controlled sequence.

Cont::rbegin:

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

Cont::rend:

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

Cont::size:

```
size_type size() const;
```

The member function returns the length of the controlled sequence, in constant time regardless of the length of the controlled sequence.

Cont::swap:

```
void swap(Cont& x);
```

The member function swaps the controlled sequences between `*this` and `x`. If `get_allocator() == x.get_allocator()`, it does so in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Functions

operator!=:

```
template<class T>
    bool operator!=(
        const Cont <T>& lhs,
        const Cont <T>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==:

```
template<class T>
    bool operator==(
        const Cont <T>& lhs,
        const Cont <T>& rhs);
```

The template function overloads `operator==` to compare two objects of template class Cont. The function returns `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end(), rhs.begin())`.

operator<:

```
template<class T>
    bool operator<(
        const Cont <T>& lhs,
        const Cont <T>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `Cont`. The function returns `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=:

```
template<class T>
    bool operator<=(
        const Cont <T>& lhs,
        const Cont <T>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>:

```
template<class T>
    bool operator*gt;(
        const Cont <T>& lhs,
        const Cont <T>& rhs);
```

The template function returns `rhs < lhs`.

operator>=:

```
template<class T>
    bool operator>=(
        const Cont <T>& lhs,
        const Cont <T>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap:

```
template<class T>
    void swap(
        Cont <T>& lhs,
        Cont <T>& rhs);
```

The template function executes `lhs.swap (rhs)`.

Notices

Dinkumware Notices

IBM Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks and service marks

IBM®, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Industry standards

The following standards are supported:

- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1990).
- The C language is also consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1999 (E)).

- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The C++ language is also consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:2003 (E)).
- The C and C++ languages are consistent with the OpenMP C and C++ Application Programming Interface Version 2.5.

References

- **ANSI Standard X3.159-1989** (New York NY: American National Standards Institute, 1989). The original C Standard, developed by the ANSI-authorized committee X3J11. The Rationale that accompanies the C Standard explains many of the decisions that went into it, if you can get your hands on a copy.
- **ISO/IEC Standard 9899:1990** (Geneva: International Standards Organization, 1990). The official C Standard around the world. Aside from formatting details and section numbering, the ISO C Standard is identical to the ANSI C Standard.
- **ISO/IEC Amendment 1 to Standard 9899:1990** (Geneva: International Standards Organization, 1995). The first (and only) amendment to the C Standard. It provides substantial support for manipulating large character sets.
- **ISO/IEC Standard 14882:1998** (Geneva: International Standards Organization, 1998). The official C++ Standard around the world. The ISO C++ Standard is identical to the ANSI C++ Standard.
- * P.J. Plauger, **The Standard C Library** (Englewood Cliffs NJ: Prentice Hall, 1992). Contains a complete implementation of the Standard C library, as well as text from the library portion of the C Standard and guidance in using the Standard C library.
- * P.J. Plauger, **The Draft Standard C++ Library** (Englewood Cliffs NJ: Prentice Hall, 1995). Contains a complete implementation of the draft Standard C++ library as of early 1994.
- P.J. Plauger, Alexander Stepanov, Meng Lee, and David R. Musser, **The Standard Template Library** (Englewood Cliffs NJ: Prentice Hall, 1999). Contains a complete implementation of the Standard Template Library as incorporated into the C++ Standard.
- **Draft Technical Report on C++ Library Extensions – TR1**. This draft technical report has been submitted to the C++ standards committee, and is available at: www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1836.pdf .

Product Number: 5650-ZOS

Printed in USA

SC14-7309-00