IBM XL C/C++ for Linux, V13.1.3

IBM

Language Reference
for Little Endian Distributions

Version 13.1.3

IBM XL C/C++ for Linux, V13.1.3

Language Reference
for Little Endian Distributions

Version 13.1.3

Contents

About this document

This document describes the syntax, semantics, and IBM® XL C/C++ for Linux implementation of the C and C++ programming language extensions. Although the XL C/C++ compiler conforms to the specifications maintained by the ISO standards for the C and C++ programming languages, the compiler also incorporates many extensions to the core languages. These extensions have been implemented to enhance usability in specific operating environments, support compatibility with other compilers, and support new hardware capabilities. For example, many language constructs have been added for compatibility with the GNU Compiler Collection (GCC), to maximize portability between the two development environments.

Note: Detailed descriptions of standard features are no longer provided in this document. Instead, a list of the standard features that the compiler currently supports is provided in Standard features. For a description of these standard features, see the C and C++ language standards.

Who should read this document

This document is a reference for users who want to learn about IBM extension features. Users can also access the list of standard features that the compiler currently supports.

How to use this document

This document contains detailed descriptions for IBM extension features. It does not include the following topics:
- Detailed descriptions of standard C and C++ features. For a description of these standard features, see the C/C++ standard.
- Standard C and C++ library functions and headers. For information on the standard C and C++ libraries, refer to your operating system information.
- Compiler pragmas, predefined macros, and built-in functions. These are described in the *XL C/C++ Compiler Reference*.

How this document is organized

- Chapter 1 provides a brief introduction to language levels and language extensions.
- Chapter 2 describes all the IBM extension features that are categorized in different groups.
- Chapter 3 lists all the standard features that the compiler currently supports.

Conventions

Typographical conventions

The following table shows the typographical conventions used in the IBM XL C/C++ for Linux, V13.1.3 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **xlc** and **xlC** (**xlc++**), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	nomaf | <u>maf</u>
`monospace`	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize myprogram.c, enter: `xlc myprogram.c -O3`.

Qualifying elements (icons)

Most features described in this information apply to both C and C++ languages. In descriptions of language elements where a feature is exclusive to one language, or where functionality differs between languages, this information uses icons to delineate segments of text as follows:

Table 2. Qualifying elements

Qualifier/Icon	Meaning
C only begins C only ends	The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.
C++ only begins C++ only ends	The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.
IBM extension begins IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.

Table 2. Qualifying elements (continued)

Qualifier/Icon	Meaning
C11 begins ▶ C11 C11 ◀ C11 ends	The text describes a feature that is introduced into standard C as part of C11.
C++11 begins ▶ C++11 C++11 ◀ C++11 ends	The text describes a feature that is introduced into standard C++ as part of C++11.
C++14 begins ▶ C++14 C++14 ◀ C++14 ends	The text describes a feature that is introduced into standard C++ as part of C++14.

Syntax diagrams

Throughout this information, diagrams illustrate XL C/C++ syntax. This section helps you to interpret and use those diagrams.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ▶▶── symbol indicates the beginning of a command, directive, or statement.

  The ──▶ symbol indicates that the command, directive, or statement syntax is continued on the next line.

  The ▶── symbol indicates that a command, directive, or statement is continued from the previous line.
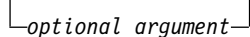
  The ──▶◀ symbol indicates the end of a command, directive, or statement.

  Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the │── symbol and end with the ──│ symbol.
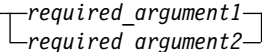
- Required items are shown on the horizontal line (the main path):

  ▶▶──keyword──*required_argument*──▶◀

- Optional items are shown below the main path:

  ▶▶──keyword──┬──────────────────┬──────────────────────────────────────▶◀
                 └─*optional_argument*─┘

- If you can choose from two or more items, they are shown vertically, in a stack.

  If you *must* choose one of the items, one item of the stack is shown on the main path.

  ▶▶──keyword──┬─*required_argument1*─┬──────────────────────────────────▶◀
                 └─*required_argument2*─┘

If choosing one of the items is optional, the entire stack is shown below the main path.

```
▶▶──keyword──┬──────────────────┬──────────────────────────▶◀
             ├─optional_argument1─┤
             └─optional_argument2─┘
```
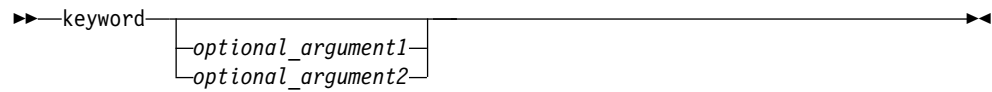
- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:

```
          ┌─,──────────────┐
▶▶──keyword──▼─repeatable_argument─┴──────────────────────────▶◀
```

- The item that is the default is shown above the main path.

```
             ┌─default_argument──┐
▶▶──keyword──┴─alternate_argument─┴──────────────────────────▶◀
```

- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Example of a syntax statement

EXAMPLE *char_constant* {*a*|*b*}[*c*|*d*]*e*[,*e*]... *name_list*{*name_list*}...

The following list explains the syntax statement:
- Enter the keyword EXAMPLE.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Optionally, enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each *name*.

Note: The same example is used in both the syntax-statement and syntax-diagram representations.

Examples in this information

The examples in this information, except where otherwise noted, are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible methods to achieve a specific result.

The examples for installation information are labelled as either *Example* or *Basic example*. *Basic examples* are intended to document a procedure as it would be performed during a basic, or default, installation; these need little or no modification.

Related information

The following sections provide related information for XL C/C++:

IBM XL C/C++ information

XL C/C++ provides product information in the following formats:

- Quick Start Guide

  The Quick Start Guide (`quickstart.pdf`) is intended to get you started with IBM XL C/C++ for Linux, V13.1.3. It is located by default in the XL C/C++ directory and in the `\quickstart` directory of the installation DVD.

- README files

  README files contain late-breaking information, including changes and corrections to the product information. README files are located by default in the XL C/C++ directory, and in the root directory and subdirectories of the installation DVD.

- Installable man pages

  Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *IBM XL C/C++ for Linux, V13.1.3 Installation Guide*.

- Online product documentation

  The fully searchable HTML-based documentation is viewable in IBM Knowledge Center at http://www.ibm.com/support/knowledgecenter/SSXVZZ_13.1.3/com.ibm.compilers.linux.doc/welcome.html.

- PDF documents

  PDF documents are available on the web at http://www.ibm.com/support/docview.wss?uid=swg27036675.

  The following files comprise the full set of XL C/C++ product information:

Table 3. XL C/C++ PDF files

Document title	PDF file name	Description
IBM XL C/C++ for Linux, V13.1.3 Installation Guide, GC27-6540-02	`install.pdf`	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C/C++ for Linux, V13.1.3, GI13-2875-02	`getstart.pdf`	Contains an introduction to the XL C/C++ product, with information about setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C/C++ for Linux, V13.1.3 Compiler Reference, SC27-6570-02	`compiler.pdf`	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions.
IBM XL C/C++ for Linux, V13.1.3 Language Reference, SC27-6550-02	`langref.pdf`	Contains information about language extensions for portability and conformance to nonproprietary standards.
IBM XL C/C++ for Linux, V13.1.3 Optimization and Programming Guide, SC27-6560-02	`proguide.pdf`	Contains information about advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization, and the XL C/C++ high-performance libraries.

To read a PDF file, use Adobe Reader. If you do not have Adobe Reader, you can download it (subject to license terms) from the Adobe website at http://www.adobe.com.

More information related to XL C/C++, including IBM Redbooks® publications, white papers, and other articles, is available on the web at http://www.ibm.com/support/docview.wss?uid=swg27036675.

For more information about C/C++, see the C/C++ café at https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=5894415f-be62-4bc0-81c5-3956e82276f3.

Standards and specifications

XL C/C++ is designed to support the following standards and specifications. You can refer to these standards and specifications for precise definitions of some of the features found in this information.

- *Information Technology - Programming languages - C, ISO/IEC 9899:1990*, also known as *C89*.
- *Information Technology - Programming languages - C, ISO/IEC 9899:1999*, also known as *C99*.
- *Information Technology - Programming languages - C, ISO/IEC 9899:2011*, also known as *C11*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:1998*, also known as *C++98*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2003*, also known as *C++03*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2011*, also known as *C++11*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2014*, also known as *C++14* (Partial support).
- *AltiVec Technology Programming Interface Manual*, Motorola Inc. This specification for vector data types, to support vector processing technology, is available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- *OpenMP Application Program Interface Version 3.1* (full support), *OpenMP Application Program Interface Version 4.0 (partial support)*, and *OpenMP Application Program Interface Version 4.5 (partial support)*, available at http://www.openmp.org

Other IBM information

- *ESSL product documentation* available at http://www.ibm.com/support/knowledgecenter/SSFHY8/essl_welcome.html?lang=en

Other information

- *Using the GNU Compiler Collection* available at http://gcc.gnu.org/onlinedocs

Technical support

Additional technical support is available from the XL C/C++ Support page at http://www.ibm.com/support/entry/portal/product/rational/xl_c/c++_for_linux. This page provides a portal with search capabilities to a large selection of Technotes and other support information.

If you cannot find what you need, you can send an email to compinfo@cn.ibm.com.

For the latest information about XL C/C++, visit the product information site at http://www.ibm.com/software/products/en/xlcpp-linux.

How to send your comments

Your feedback is important in helping us to provide accurate and high-quality information. If you have any comments about this information or any other XL C/C++ information, send your comments to compinfo@cn.ibm.com.

Be sure to include the name of the manual, the part number of the manual, the version of XL C/C++, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

Chapter 1. Language levels and language extensions

The C and C++ languages described in this reference are based on the standards listed in "Standards and specifications" on page x.

We refer to the following language specifications as "base language levels" in order to introduce the notion of an extension to a base.

- C++11
- C++03
- C++98
- C11
- C99
- C89

This information uses the term *K&R C* to refer to the C language plus the generally accepted extensions produced by Brian Kernighan and Dennis Ritchie that were in use prior to the ISO standardization of C.

Note: Detailed descriptions of standard features are no longer provided in the Language Reference. Instead, a list of the standard features that the compiler currently supports is provided in Standard features. For a description of these standard features, see the C and C++ language standards.

In addition to the features supported by the base levels, XL C/C++ contains language extensions that enhance usability and facilitate porting programs to different platforms, including:

- "Extensions to C++ to support C99 and C11 standard features" on page 2
- "Extensions related to GNU C and GNU C++" on page 2
- "Extensions supporting and extending the AltiVec Programming Interface" on page 2

You can control the language level to be used for compilation through several mechanisms, including:

- various invocation commands in the *XL C/C++ Compiler Reference*
- the **-std (-qlanglvl)** option in the *XL C/C++ Compiler Reference*

With a few exceptions, almost all of the language extensions are supported when you compile using the basic invocation commands **xlc** (for C) and **xlc++** or **xlC** (for C++).

The default language level for the **xlc** invocation command is **extc99**, which includes all of the features introduced by the C99 standard, and most of the IBM extensions described in this information.

The default language level for the **xlC** or **xlc++** invocation command is **extended**, which includes most of the IBM extensions described in this information, as well as many C99 features.

1

For information on the various methods for controlling the language level for compilation, see Invoking the compiler in the *XL C/C++ Compiler Reference* and **-std (-qlanglvl)** in the *XL C/C++ Compiler Reference*.

Extensions to C++ to support C99 and C11 standard features

The C++ language specification does not include many of the features specified in the C99 and C11 language standards. To promote compatibility and portability between C++ and C99 or between C++ and C11, the XL C++ compiler enables many of the C99 and C11 features that are supported by the XL C compiler. Because these features extend the C++ standard, they are considered extensions to the base language. In this reference, unless the text is marked to indicate that a feature is supported only in C, C99, or C11, C99 and C11 features also apply to C++. A complete list of C99 and C11 features supported in XL C++ is provided in "Extensions for C99 compatibility" on page 50 and "Extensions for C11 compatibility" on page 51.

Extensions related to GNU C and GNU C++

Certain language extensions that correspond to GNU C and GNU C++ features are implemented to facilitate portability. These include extensions to C89, C99, C++98, C++03, and C++11. Throughout this information, the text indicates the IBM extensions that have been implemented for compatibility with GNU C and GNU C++; a complete list of these is provided in "Extensions for GNU C compatibility" on page 6 for both C and C++, "Extensions for GNU C compatibility" on page 48 for C only, and "Extensions for GNU C++ compatibility" on page 51 for C++ only.

Extensions supporting and extending the AltiVec Programming Interface

XL C/C++ supports and extends AltiVec vector types when vector support is enabled. These language extensions exploit the SIMD capabilities of the PowerPC® processor, and facilitate the associated optimization techniques. The IBM implementation of the AltiVec Programming Interface specification is an extended implementation, which, for the most part, matches the syntax and semantics of the GNU C implementation. In addition to the text provided throughout this information that describes the behavior of the vector extensions, a list of the IBM extensions to the AltiVec Programming Interface is also provided in "Extensions for vector processing support" on page 39.

Chapter 2. IBM extension features

This chapter describes features that are IBM extensions to the standard language specifications.

Note: In a topic that describes both standard language elements and IBM extension features, the IBM extension information is indicated in one of the following ways. If there are none of these indicators in the topic, the entire topic describes an IBM extension.

- Enclosed in icons ▶ IBM and IBM ◀
- Marked with (IBM extension) in the section title
- Specified as IBM extension in main text

IBM extension features for both C and C++

This section describes IBM extension features for both the C and C++ languages that are listed in the following categories. The topic of the extension feature is appended under only one category if it belongs to more than one category; in the other categories that the feature belongs to, only a link to the feature is provided.

- "General IBM extensions"
- "Extensions for GNU C compatibility" on page 6
- "Extensions for vector processing support" on page 39

General IBM extensions

The following feature is disabled by default at all language levels. It also can be enabled or disabled by an individual option.

Table 4. General IBM extensions with individual option controls

Language feature	Discussed in:	Individual option controls
Extra text after #endif or #else	"Extension of #endif and #else" on page 4	**-Wno-extra-tokens**

▶ C The following feature is enabled by default with the xlc, cc and c99 invocation commands when the **extc99**, **stdc99**, **extc1x**, or **stdc11** language level is not in effect. C ◀

▶ C++ The following feature is enabled with the **-qlanglvl=extended** option. C++ ◀

Table 5. General IBM extensions

Language feature	Discussed in:
Non-C99 IBM `long long` extension	"Types of integer literals that are supported in pre-C99 and pre-C++11 modes" on page 4

▶ C The following feature is enabled by default with the xlc, cc and c99 invocation commands when the **extc99**, **stdc99**, **extc1x**, or **stdc11** language level is in effect. C ◀

> **C++** The following feature is enabled with the **-qlanglvl=extended0x** or **-qlanglvl=extended1y** option. **C++** ◄

Table 6. General IBM extensions

Language feature	Discussed in:
C99 `long long` feature with the associated IBM extensions	Types of integer literals in C99 and C++11

Extension of #endif and #else

The C and C++ language standards do not support extra text after `#endif` or `#else`. XL C/C++ compiler complies with the standards. When you port code from a compiler that supports extra text after `#endif` or `#else`, you can specify option **-Wno-extra-tokens** to suppress the warning message that is emitted.

One use is to comment on what is being tested by the corresponding `#if` or `#ifdef`. For example:

```
#ifdef MY_MACRO
...
#else MY_MACRO not defined
...
#endif MY_MACRO
```

In this case, if you want the compiler to be silent about this deviation from the standards, you can suppress the message by specifying option **-Wno-extra-tokens**.

Integer literals

The `long long` features

There are two `long long` features:
- the C99 `long long` feature
- the non-C99 `long long` feature

Note: The syntax of integer literals is the same for both of the `long long` features.

Types of integer literals that are supported in pre-C99 and pre-C++11 modes

The following table lists the integer literals and shows the possible data types when the C99 `long long` feature is not enabled.

Table 7. Types of integer literals that are supported in pre-C99 and pre-C++11 modes[1]

Representation	Suffix	Possible data types					
		int	unsigned int	long int	unsigned long int	► IBM long long int	► IBM unsigned long long int
Decimal	None	+		+	$+^2$		
Octal, Hex	None	+	+	+	+		
All	u or U		+		+		
Decimal	l or L			+	+		
Octal, Hex	l or L			+	+		

4   XL C/C++: Language Reference for Little Endian Distributions

Table 7. Types of integer literals that are supported in pre-C99 and pre-C++11 modes[1] (continued)

Representation	Suffix	Possible data types					
All	Both u or U and l or L				+		
Decimal	ll or LL					+	+
Octal, Hex	ll or LL					+	+
All	Both u or U and ll or LL						+

Note:

1. When none of the long long features are enabled, types of integer literals include all the types in this table except the last two columns.

2. ➤ **IBM** The unsigned long int type is not required here in the C++98 and C++03 standards. The C++ compiler includes the type in the implementation for compatibility purposes only.

Types of integer literals that are supported in C99 and C++11

When both the C99 and non-C99 long long features are disabled, integer literals that have one of the following suffixes cause a severe compile-time error:

- ll or LL
- Both u or U and ll or LL

➤ **C++11** To strictly conform to the C++11 standard, the compiler introduces the extended integer safe behavior to ensure that a signed value never becomes an unsigned value after a promotion. After you enable this behavior, if a decimal integer literal that does not have a suffix containing u or U cannot be represented by the long long int type, the compiler issues a warning message to indicate that the value of the literal is out of range. You can change the severity of the message from warning to error with the **-Werror=implicitly-unsigned-literal** option. The extended integer safe behavior is the only difference between the C99 long long feature with the associated IBM extensions and the C99 long long feature. **C++11** ◄

The following table lists the integer literals and shows the possible data types when the C99 long long feature is enabled.

Table 8. Types of integer literals that are supported in C99 and C++11

Representation	Suffix	Possible data types					
		int	unsigned int	long int	unsigned long int	long long int	unsigned long long int
Decimal	None	+		+			+[1,2]
Octal, Hex	None	+	+	+	+		
All	u or U		+		+		
Decimal	l or L			+			+[1,2]

Table 8. Types of integer literals that are supported in C99 and C++11 (continued)

Representation	Suffix	Possible data types					
Octal, Hex	l or L			+	+		
All	Both u or U and l or L				+		
Decimal	ll or LL					+	+[1, 2]
Octal, Hex	ll or LL					+	+
All	Both u or U and ll or LL						+

Note:

1. ▶ `C++11` The compiler does not support this type if the extended integer safe behavior is enabled.

2. ▶ `IBM` All integer literals can be represented by the `unsigned long long int` type if they can fit into this type. A decimal literal without a u or U in the suffix is represented by the `unsigned long long int` type if both of the following conditions are satisfied. In this case, the compiler generates a message to indicate that the value of the literal is too large for any signed integer type.
   - The value of the literal can fit into the `unsigned long long int` type.
   - The value cannot fit into any of the possible data types that are not marked as an IBM extension in the table.

3. The C/C++ standard defines the following rules:
   - When `long int` is a possible data type, `long long int` is also a possible data type.
   - When `unsigned long int` is a possible data type, `unsigned long long int` is also a possible data type.

   However, in 64-bit mode, the additional cases that are defined by the rules are never encountered.

Related reference:

📄 See -std (-qlanglvl) in the XL C/C++ Compiler Reference

Extensions for GNU C compatibility

The following features are enabled by default at all language levels:

Table 9. Default IBM XL C and C++ extensions for GNU C compatibility

Language feature	Discussed in:
`__alignof__` operator	N/A
`__attribute__` keyword	"Variable attributes" on page 32, "Function attributes" on page 9
`__complex__` keyword	N/A
`__extension__` keyword	N/A
`__imag__` and `__real__` complex type operators	N/A
`__restrict__` keyword	N/A
`__thread` storage class specifier	"The __thread storage class specifier" on page 28
`__typeof__` keyword	"The typeof keyword" on page 29
`#include_next` preprocessor directive	"The #include_next directive" on page 27

Language feature	Discussed in:
`#warning` preprocessor directive	N/A
Alternate keywords	N/A
`asm` inline assembly-language statements	"Inline assembly statements" on page 19
`asm` labels	N/A
Complex literal suffixes	N/A
Computed `goto` statements	N/A
Dollar signs in identifiers	"Characters in identifiers" on page 8
FORTIFY_SOURCE	N/A
Function attributes	"Function attributes" on page 9
Initialization of static variables by compound literals	"Compound literal expressions" on page 8
Labels as values	N/A
Postfix and unary operators on complex types (increment, decrement, and complex conjugation)	N/A
Statements and declarations in expressions (statement expressions)	N/A
Static initialization of flexible array members of aggregates	"Flexible array members of structures" on page 8
Structures with flexible array members being members of another structure	"Flexible array members of structures" on page 8
Type attributes	"Type attributes" on page 29
Variable attributes	"Variable attributes" on page 32
Variadic macro extensions	"Variadic macros" on page 38
Zero-extent arrays	N/A

▶ **C** The following features are enabled by default when you compile with the **xlc** invocation command or the **-qlanglvl=extc99 | extc89 | extc1x | extended** options. **C** ◀

▶ **C++** The following features are enabled by default at all C++ language levels: **C++** ◀

Some features can also be enabled or disabled by specific compiler options, which are listed in the below table:

Table 10. IBM XL C and C++ extensions for GNU C compatibility with individual option controls

Language feature	Discussed in:	Individual option controls
`typeof` keyword	"The `typeof` keyword" on page 29	N/A
Visibility function attribute[1]	"visibility" on page 18	**-fvisibility (-qvisibility)**
Visibility variable attribute[1]	"The visibility variable attribute" on page 37	**-fvisibility (-qvisibility)**

Table 10. IBM XL C and C++ extensions for GNU C compatibility with individual option controls (continued)

Language feature	Discussed in:	Individual option controls
Note:		
1. You can use the **-fvisibility** option to specify visibility attributes for variables and functions if they do not get visibility attributes from pragma directives, explicitly specified attributes, or propagation rules. This option cannot be used to disable visibility attributes for variables or functions.		

Characters in identifiers

The dollar sign can appear in identifier names at all language levels. When you compile with either the **-fno-dollars-in-identifiers** or the **-qnodollar** option, the compiler issues error messages for dollar signs in identifier names.

Other specialized identifiers, such as characters in national character sets, can also be allowed to appear in an identifier depending on compiler options.

Compound literal expressions

For compatibility with GNU C, a static variable can be initialized with a compound literal of the same type, provided that all the initializers in the initializer list are constant expressions.

Flexible array members of structures

To be compatible with GNU C/C++, the XL C/C++ compiler extends Standard C and C++, to ease the restrictions on flexible array members and allow the following situations:

- Structures containing flexible array members can be members of other structures.
- C Flexible array members can be statically initialized only if either of the following two conditions is true:
  - The flexible array member is the last member of the structure, for example:

    ```
    struct f {
      int a;
      int b[];
    } f1 = {1,{1,2,3}}; // Fine.

    struct a {
      int b;
      int c[];
      int d[];
    } e = { 1,{1,2},3}; // Error, c is not the last member
                        // of structure a.
    ```

  - Flexible array members are contained in the outermost structure of nested structures. Members of inner structures cannot be statically initialized, for example:

    ```
    struct b {
      int c;
      int d[];
    };

    struct c {
      struct b f;
      int g[];
    } h ={{1,{1,2}},{1,2}}; // Error, member d of structure b is
                            // in the inner nested structure.
    ```

C ◄

Function attributes

Function attributes are extensions implemented to enhance the portability of programs developed with GNU C. Specifiable attributes for functions provide explicit ways to help the compiler optimize function calls and to instruct it to check more aspects of the code. Others provide additional functionality.

IBM XL C/C++ compiler implements a subset of the GNU C function attributes. For a particular function attribute that is not implemented, the compiler issues diagnostics and ignores the attribute specification.

A function attribute is specified with the keyword __attribute__ followed by the attribute name and any additional arguments the attribute name requires. A function __attribute__ specification is included in the declaration or definition of a function. The syntax takes the following forms:

Function attribute syntax: function definition (form 1)
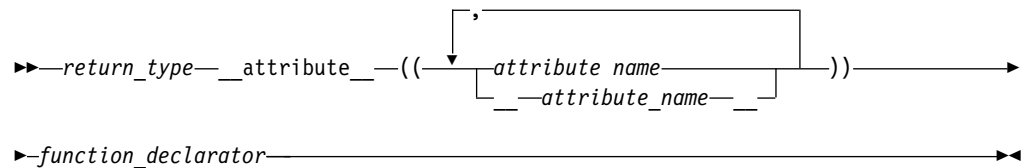
```
►►─return_type─__attribute__─((─┬─▼─┬─attribute name─────┬─))──────────►
                                │   └─__─attribute_name─__─┘
                                └────────,───────────────────

►─function_declarator──────────────────────────────────────────────►◄
```
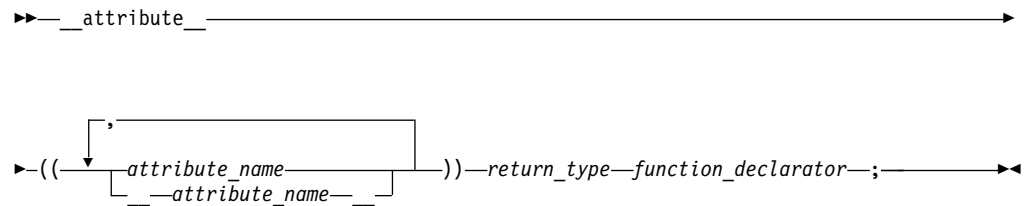
Function attribute syntax: function definition (form 2)

```
►►──__attribute__───────────────────────────────────────────────────►

►─((─┬─▼─┬─attribute_name────┬─))─return_type─function_declarator─;──►◄
     │   └─__─attribute_name─__─┘
     └──────,─────────────────
```

Function attribute syntax: function definition (form 3)

```
►►─return_type─function_declarator──__attribute__────────────────────►

►─((─┬─▼─┬─attribute_name────┬─))──;────────────────────────────────►◄
     │   └─__─attribute_name─__─┘
     └──────,─────────────────
```

You can specify *attribute_name* with or without leading and trailing double underscore characters; however, using the double underscore characters reduces the likelihood of name conflicts with macros of the same name. These language features are collectively available when compiling in any of the extended language levels.

The following function declarations are all valid:

```
int __attribute__((attribute_name)) func(int i);   //Form 1
__attribute__((attribute_name)) int func(int);     //Form 2
int func() __attribute__((attribute_name));        //Form 3
```

The following function attributes are supported:
- "alias"
- "always_inline" on page 11
- "const" on page 11
- "constructor and destructor" on page 12
- "format" on page 12
- "format_arg" on page 13
- "gnu_inline" on page 14
- "malloc" on page 15
- "noinline" on page 16
- "noreturn" on page 16
- "pure" on page 16
- "The section function attribute" on page 16
- "used" on page 17
- "weak" on page 17
- "weakref" on page 17
- "visibility" on page 18

Related reference:

"Variable attributes" on page 32

"Type attributes" on page 29

alias:
The alias function attribute causes the function declaration to appear in the object file as an alias for another symbol. This language feature provides a technique for coping with duplicate or cumbersome names.

alias function attribute syntax

►►— __attribute__ —((—┬─alias──────┬─(—"original_function_name"—)—))——————►◄
                       └─__alias__──┘

► C ◄ The aliased function can be defined after the specification of its alias with this function attribute. C also allows an alias specification in the absence of a definition of the aliased function in the same compilation unit.

The following code declares func1 to be an alias for __func2:

```
void __func2(){   /* function body */   }
void func1() __attribute__((alias("__func2")));
```

C ◄

► C++ ◄ The *original_function_name* must be the mangled name.

The following code declares func1 to be an alias for __func2

```
extern "C" __func2(){   /*  function body  */   }
void func1() __attribute__((alias("__func2")));
```

C++

The compiler does not check for consistency between the declaration of func1 and definition of __func2. Such consistency remains the responsibility of the programmer.

Related reference:
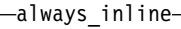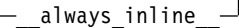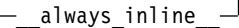
"The weak variable attribute" on page 37

always_inline:
The always_inline function attribute instructs the compiler to inline a function. This function can be inlined when all of the following conditions are satisfied:

- The function is an inline function that satisfies any of the following conditions:
    - The function is specified with the inline or __inline__ keyword.
    - The option **-qinline+<function_name>** is specified, where *function_name* is the name of the function to be inlined.
    - ▶ C++ The function is defined within a class declaration. C++ ◀
- The function is not specified with the noinline or __noinline__ attribute.
- The number of functions to be inlined does not exceed the limit of inline functions that can be supported by the compiler.

always_inline function attribute syntax

```
►►──__attribute__──((──┬─always_inline───────┬──))──────────────────────►◄
                       └─__always_inline__──┘
```

The noinline attribute takes precedence over the always_inline attribute. The always_inline attribute takes precedence over inlining compiler options only if inlining is enabled. The always_inline attribute is ignored if inlining is disabled.

▶ C++ The compiler might not inline a virtual function even when the function is specified with the always_inline attribute. The compiler will not issue an informational message to indicate that a virtual function is not inlined.

When you specialize a function template that is specified with the always_inline attribute, this attribute is propagated to the template specification. If you apply the always_inline attribute to the template specification, the duplicate always_inline attribute is ignored. See the following example.

```
template<class T> inline __attribute__((always_inline)) T test( ){
    return (T)0;
}

// The duplicate attribute "always_inline" is ignored.
template<> inline __attribute__((always_inline)) float test<float>(){
    return (float)0;
}
```

C++

Related reference:

"noinline" on page 16

const:

The const function attribute allows you to tell the compiler that the function can safely be called fewer times than indicated in the source code. The language feature provides you with an explicit way to help the compiler optimize code by indicating that the function does not examine any values except its arguments and has no effects except for its return value.

const function attribute syntax

```
►►──__attribute__──((──┬──const────┬──))──────────────────────────►◄
                       └──__const__─┘
```

The following kinds of functions should not be declared const:
- A function with pointer arguments which examines the data pointed to.
- A function that calls a non-const function.

Note: GNU C has a non-attribute method that uses the const keyword to achieve the const function attribute, but the XL C/C++ compiler does not support this method.

Related reference:

See -qisolated_call in the XL C/C++ Compiler Reference

constructor and destructor:
The constructor and destructor function attributes provide the ability to write a function that initializes data or releases storage that is used implicitly during program execution. A function to which the constructor function attribute has been applied is called automatically before execution enters main. Similarly, a function to which the destructor attribute has been applied is called automatically after calling exit or upon completion of main.

When the constructor or destructor function is called automatically, the return value of the function is ignored, and any parameters of the function are undefined.

constructor and destructor function attribute syntax

```
►►──__attribute__──((──┬──constructor────┬──))──────────────────────────►◄
                       ├──destructor──────┤
                       ├──__constructor__─┤
                       └──__destructor__──┘
```

A function declaration containing a constructor or destructor function attribute must match all of its other declarations.

format:
The format function attribute provides a way to identify user-defined functions that take format strings as arguments so that calls to these functions will be type-checked against a format string, similar to the way the compiler checks calls to the functions printf, scanf, strftime, and strfmon for errors.

format function attribute syntax

```
                                    ,
                            ┌───────◄───────┐
►►── __attribute__ ──(( ───┬─format───┬──(─┬─printf────┬──,─string_index─,─first_to_check─)─┘──)) ──────────►◄
                           └─__format__┘    ├─scanf─────┤
                                            ├─strftime──┤
                                            ├─strfmon───┤
                                            ├─__printf__─┤
                                            ├─__scanf__──┤
                                            ├─__strftime__─┤
                                            └─__strfmon__─┘
```
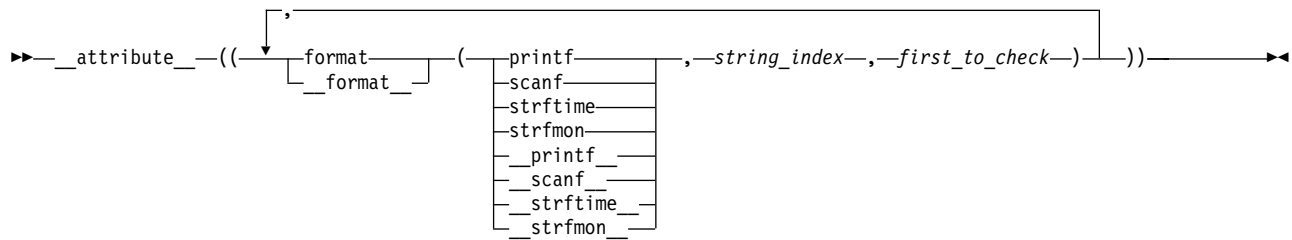
where

string_index

> Is a constant integral expression that specifies which argument in the declaration of the user function is the format string argument. ► C++ ◄ In C++, the minimum value of *string_index* for nonstatic member functions is 2 because the first argument is an implicit `this` argument. This behavior is consistent with that of GNU C++. C++ ◄

first_to_check

> Is a constant integral expression that specifies the first argument to check against the format string. If there are no arguments to check against the format string (that is, diagnostics should only be performed on the format string syntax and semantics), *first_to_check* should have a value of 0. For `strftime`-style formats, *first_to_check* is required to be 0.

It is possible to specify multiple `format` attributes on the same function, in which case, all apply.

```
void my_fn(const char* a, const char* b, ...)
    __attribute__((__format__(__printf__,1,0), __format__(__scanf__,2,3)));
```

It is also possible to diagnose the same string for different format styles. All styles are diagnosed.

```
void my_fn(const char* a, const char* b, ...)
    __attribute__((__format__(__printf__,2,3),
                   __format__(__strftime__,2,0),
                   __format__(__scanf__,2,3)));
```

format_arg:
The `format_arg` function attribute provides a way to identify user-defined functions that modify format strings. Once the function is identified, calls to functions like `printf`, `scanf`, `strftime`, or `strfmon`, whose operands are a call to the user-defined function can be checked for errors.

format_arg function attribute syntax

```
►►── __attribute__ ──(( ──┬─format_arg───┬──(─string_index─)─)) ──────────────►◄
                          └─__format_arg__─┘
```
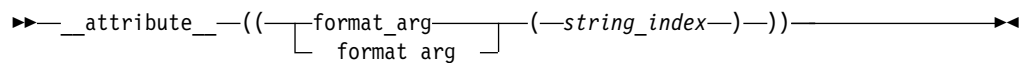
where *string_index* is a constant integral expression that specifies which argument is the format string argument, starting from 1. ► C++ ◄ For non-static member functions in C++, *string_index* starts from 2 because the first parameter is an implicit `this` parameter. C++ ◄

It is possible to specify multiple `format_arg` attributes on the same function, in which case, all apply.

```
extern char* my_dgettext(const char* my_format, const char* my_format2)
        __attribute__((__format_arg__(1))) __attribute__((__format_arg__(2)));

printf(my_dgettext("%","%"));
//printf-style format diagnostics are performed on both "%" strings
```

gnu_inline: The `gnu_inline` attribute instructs the compiler to modify the inlining behavior of a function. When this function attribute is used, the compiler imitates the GNU legacy inlining extension to C.

This function attribute is only enabled if used in conjunction with an inline keyword (`__inline__`, `inline`, `__inline`, etc.).

gnu_inline function attribute syntax

►►—inline—`__attribute__`—((——gnu_inline——))——————————————————►◄

Note: The behavior of the `gnu_inline` function attribute is the same when used in conjunction with either the `inline` or `__inline__` keywords.

The semantics of the GNU legacy inlining extension to C are as follows:

▶ C

extern gnu_inline:
```
extern inline __attribute__((gnu_inline)) func() {...};
```

This definition of `func` is used only for inlining. It is not compiled as a standalone function.

static gnu_inline:
```
static inline __attribute__((gnu_inline)) func() {...};
```

If the function is generated, it is generated with internal linkage.

plain gnu_inline:
```
inline __attribute__((gnu_inline)) func() {...};
```

The definition is used for inlining when possible. It is compiled as a standalone function (emitted as a strong definition) and emitted with external linkage.

C ◀

▶ C++

extern gnu_inline:
```
[extern] inline __attribute__((gnu_inline)) func() {...};
```

This definition of `func` is used only for inlining. It is not compiled as a standalone function. Note that member functions (including static ones and ones with no linkage) marked with function attribute `gnu_inline` has "extern" behavior.

static gnu_inline:
```
static inline __attribute__((gnu_inline)) func() {...};
```

If the function is generated, it is generated with internal linkage. Note that static behavior only applies to non-member static functions.

C++

The `gnu_inline` attribute can be specified inside double parentheses with keyword `__attribute__` in a function declaration. See the following example.

```
inline int func() __attribute__((gnu_inline));
```

As with other GCC function attributes, the double underscores on the attribute name are optional. The `gnu_inline` attribute should be used with a function that is also declared with the `inline` keyword.

malloc: With the function attribute `malloc`, you can instruct the compiler to treat a function as if any non-NULL pointer it returns cannot alias any other valid pointers. This type of function (such as `malloc` and `calloc`) has this property, hence the name of the attribute. As with all supported attributes, `malloc` will be accepted by the compiler without requiring any particular option or language level.

The `malloc` function attribute can be specified inside double parentheses via keyword __attribute__ in a function declaration.

malloc function attribute syntax

▶▶──__attribute__──((──┬──malloc───┬──))────────────────────────▶◀
                       └──__malloc__─┘

As with other GCC function attributes, the double underscores on the attribute name are optional.

Note:
- Do not use this function attribute unless you are sure that the pointer returned by a function points to unique storage. Otherwise, optimizations performed might lead to incorrect behavior at run time.
- If the function does not return a pointer or C++ reference return type but it is marked with the function attribute `malloc`, a warning is emitted, and the attribute is ignored.

Example

A simple case that should be optimized when attribute `malloc` is used:

```
#include <stdlib.h>
#include <stdio.h>
int a;
void* my_malloc(int size)  __attribute__ ((__malloc__))
{
  void* p = malloc(size);
  if (!p) {
    printf("my_malloc: out of memory!\n");
    exit(1);
  }
  return p;
}
int main() {
  int* x = &a;
  int* p = (int*) my_malloc(sizeof(int));
  *x = 0;
```

```
    *p = 1;
    if (*x) printf("This printf statement to be detected as unreachable
                    and discarded during compilation process\n");
    return 0;
}
```

noinline:
The noinline function attribute prevents the function to which it is applied from being inlined, regardless of whether the function is declared inline or non-inline. The attribute takes precedence over inlining compiler options, the inline keyword, and the always_inline function attribute.

noinline function attribute syntax

```
►►──__attribute__──((──┬─noinline────┬──))─────────────────────────────────►◄
                       └─__noinline__─┘
```

Other than preventing inlining, the attribute does not remove the semantics of inline functions.

noreturn:
The noreturn function attribute allows you to indicate to the compiler that the function will not return the control to its caller. The language feature provides the programmer with another explicit way to help the compiler optimize code and to reduce false warnings for uninitialized variables.

The return type of the function should be void.

noreturn function attribute syntax

```
►►──__attribute__──((──┬─noreturn────┬──))─────────────────────────────────►◄
                       └─__noreturn__─┘
```

Registers saved by the calling function may not necessarily be restored before calling the nonreturning function.

pure:
The pure function attribute allows you to declare a function that can be called fewer times than what is literally in the source code. Declaring a function with the attribute pure indicates that the function has no effect except a return value that depends only on the parameters, global variables, or both.

pure function attribute syntax

```
►►──__attribute__──((──┬─pure────┬──))─────────────────────────────────────►◄
                       └─__pure__─┘
```

Related reference:

🗎 See -qisolated_call in the XL C/C++ Compiler Reference

The section function attribute:
The section function attribute specifies the section in the object file in which the compiler should place its generated code. The language feature provides the ability to control the section in which a function should appear.

section function attribute syntax

```
►►──__attribute__─((──┬─section─────┬──(──"section_name"──)──))──────────────►◄
                      └─__section__─┘
```

where *section_name* is a string literal.

Each defined function can reside in only one section. The section indicated in a function definition should match that in any previous declaration. The section indicated in a function definition cannot be overwritten, whereas one in a function declaration can be overwritten by a later specification. Moreover, if a section attribute is applied to a function declaration, the function will be placed in the specified section only if it is defined in the same compilation unit.

Related reference:

"The section variable attribute" on page 36

used: When a function is referenced only in inline assembly, you can use the `used` function attribute to instruct the compiler to emit the code for the function even if it appears that the function is not referenced.

The `used` function attribute can be specified inside double parentheses via keyword __attribute__ in a function declaration, for example, `int foo() __attribute__ ((__used__));` As with other GCC function attributes, the double underscores on the attribute name are optional.

used function attribute syntax

```
►►──__attribute__─((──┬─used─────┬──))──────────────────────────────────────►◄
                      └─__used__─┘
```

If the function attribute `gnu_inline` is specified in such a way that the function is discarded, and is specified together with the function attribute `used`, the `gnu_inline` attribute wins, and the function definition is discarded.

weak:
The `weak` function attribute causes the symbol resulting from the function declaration to appear in the object file as a weak symbol, rather than a global one. The language feature provides the programmer writing library functions with a way to allow function definitions in user code to override the library function declaration without causing duplicate name errors.

weak function attribute syntax

```
►►──__attribute__─((──┬─weak─────┬──))──────────────────────────────────────►◄
                      └─__weak__─┘
```

Related reference:
"alias" on page 10

weakref: `weakref` is an attribute attached to function declarations which must specify a target name. The target name might also be specified through the attribute `alias` in any declaration of the function.

References to the weakref function are converted into references of the target name. If the target name is not defined in the current translation unit and it is not

referenced directly or otherwise in a way that requires a definition of the target, for example if it is only referenced by using weakref functions, the reference is weak. In the presence of a definition of the target in the current translation unit, references to a weakref function resolve directly to said definition. The weakref attribute does not otherwise affect definitions of the target. A weakref function must have internal linkage.

The `weakref` attribute, as with other GCC attributes, can be expressed in a pre-fix or post-fix syntax:

pre-fix syntax
```
static __attribute__((weakref("bar"))) void foo(void);
```

post-fix syntax
```
static void foo(void) __attribute__((weakref("bar")));
```

Functions with `weakref` or `alias` attributes may refer to other such functions. The name referred to is that of the last, i.e., non-weakref and non-alias target.

Rules

If a weakref function is declared without the keyword `static`, an error message is emitted when the compiler is configured with GCC version 4.3 or later.

The target name specified in the weakref function declaration cannot directly or indirectly point to itself.

Using the `weakref` attribute without providing a target name is not recommended.

If a body is provided in a weakref function declaration with a pre-fix syntax, the attribute is ignored. A warning message reporting this situation will be emitted.

Examples

The following examples illustrates various declarations of `weakref` functions:
```
static void foo() __attribute__((weakref("bar")));
void foo() __attribute__((weakref("bar")));
static void foo() __attribute__((weakref,alias("bar")));
static void foo() __attribute__((alias("bar"),weakref));
```

visibility:
The visibility function attributes describe whether and how a function defined in one module can be referenced or used in other modules. By using this feature, you can make a shared library smaller and decrease the possibility of symbol collision. For details, see Using visibility attributes in the XL C/C++ Optimization and Programming Guide.

visibility function attribute syntax

```
►►──__attribute__──((──┬─visibility───────┬──(──┬──"default"────┬──)──))───────────────►◄
                       └─__visibility__───┘      ├──"protected"──┤
                                                 ├──"hidden"─────┤
                                                 └──"internal"───┘
```

Example

In the following example, the visibility attribute of function `void f(int i, int j)` is hidden:

```
void __attribute__((visibility("hidden"))) f(int i, int j);
```

Related reference:

See Using visibility attributes in the XL C/C++ Optimization and Programming Guide

See -fivisibility (-qvisibility) in the XL C/C++ Compiler Reference

See -shared (-qmkshrobj) in the XL C/C++ Compiler Reference

See #pragma GCC visibility push (visibility), #pragma GCC visibility pop in the XL C/C++ Compiler Reference

Inline assembly statements

Under extended language levels, the compiler provides support for embedded assembly code fragments among C and C++ source statements. This extension has been implemented for use in general system programming code, and in the operating system kernel and device drivers, which were originally developed with GNU C.

The keyword `asm` stands for assembly code. When strict language levels are used in compilation, the C compiler treats `asm` as a regular identifier and reserves `__asm` and `__asm__` as keywords. The C++ compiler always recognizes the `asm`, `__asm`, and `__asm__` keywords.

The syntax is as follows:

asm statement syntax — statement in local scope

```
►►─┬─asm────┬──┬──────────┬───────────────────────────────────────────────►
   ├─__asm──┤  └─volatile─┘
   └─__asm__┘

►─(─code_format_string─┬──────────────────────────────────────────┬─)──►◄
                       └─:─┬────────┬──┬──────────────────────────┐
                           └─output─┘  └─:─┬───────┬──┬──────────┐
                                           └─input─┘  └─:─┬──────────┐
                                                          └─clobbers─┘
```

input:

```
  ┌─,─────────────────────────────────────┐
│ ▼ ┬──────────┬──constraint─(─C_expression─)─┴──────────────────│
    └─modifier─┘
```

output:

```
          ┌──,──────────────────────────────┐
       ┌───▼─modifier─constraint─(─C_expression─)─┴───────────────────┐
├──────┴──────────────────────────────────────────────┴──────────────────┤
```
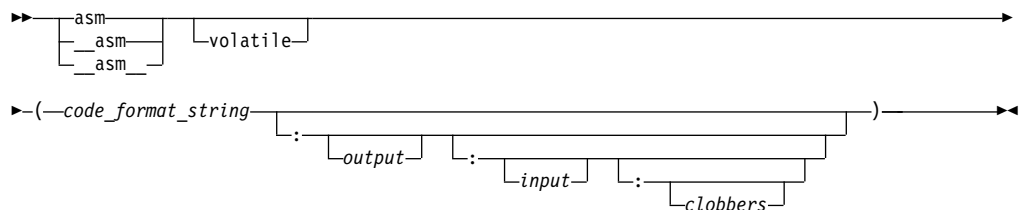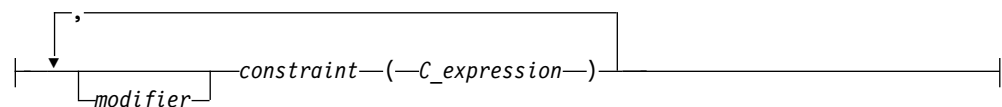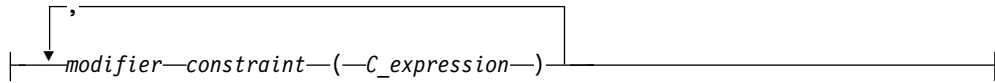
volatile

>The qualifier `volatile` instructs the compiler to perform only minimal optimizations on the assembly block. The compiler cannot move any instructions across the implicit fences surrounding the assembly block. See Example 1 for detailed usage information.

code_format_string

>The *code_format_string* is the source text of the `asm` instructions and is a string literal similar to a `printf` format specifier.

>Operands are referred to in the *%integer* format, where *integer* refers to the sequential number of the input or output operand. See Example 1 for detailed usage information.

>To increase readability, each operand can be given a symbolic name enclosed in brackets. In the assembler code section, you can refer to each operand in the *%[symbolic_name]* format, where the *symbolic_name* is referenced in the operand list. You can use any name, including existing C or C++ symbols, because the symbolic names have no relation to any C or C++ identifiers. However, no two operands in the same assembly statement can use the same symbolic name. See Example 2 for detailed usage information.

output

>The *output* consists of zero, one or more output operands, separated by commas. Each operand consists of a *constraint(C_expression)* pair. The output operand must be constrained by the = or + modifier (described below), and, optionally, by an additional % or & modifier.

input The *input* consists of zero, one or more input operands, separated by commas. Each operand consists of a *constraint(C_expression)* pair.

clobbers

>*clobbers* is a comma-separated list of register names enclosed in double quotes. If an `asm` instruction updates registers that are not listed in the *input* or *output* of the `asm` statement, the registers must be listed as clobbered registers. The following register names are valid :

>**r0 to r31**
>>General purpose registers

>**f0 to f31**
>>Floating-point registers

>**lr** Link register

>**ctr** Loop count, decrement and branching register

>**fpscr** Floating-point status and control register

>**xer** Fixed-point exception register

>**cr0 to cr7**
>>Condition registers. Example 3 shows a typical use of condition registers in the *clobbers*.

>**v0 to v31**
>>Vector registers (on selected processors only)

In addition to the register names, `cc` and `memory` can also be used in the list of clobbered registers. The usage information of `cc` and `memory` is listed as follows:

cc   Add `cc` to the list of clobbered registers if assembler instructions can alter `cr0`.

memory

> Add `memory` to the clobber list if assembler instructions can change a memory location in an unpredictable fashion. The `memory` clobber ensures that the compiler does not to move the assembler instruction across other memory references and ensures that any data that is used after the completion of the assembly statement is valid.

> However, the `memory` clobber can result in many unnecessary reloads, reducing the benefits of hardware prefetching. Thus, the `memory` clobber can impose a performance penalty and should be used with caution. See Example 4 and Example 1 for the detailed usage information.

modifier

> The *modifier* can be one of the following operators:

> **=**   Indicates that the operand is write-only for this instruction. The previous value is discarded and replaced by output data. See Example 5 for detailed usage information.

> **+**   Indicates that the operand is both read and written by the instruction. See Example 6 for detailed usage information.

> **&**   Indicates that the operand may be modified before the instruction is finished using the input operands; a register that is used as input should not be reused here.

> **%**   Declares the instruction to be commutative for this operand and the following operand. This means that the order of this operand and the next may be swapped when generating the instruction. This modifier can be used on an input or output operand, but cannot be specified on the last operand. See Example 7 for detailed usage information.

constraint

> The *constraint* is a string literal that describes the kind of operand that is permitted, one character per constraint. The following constraints are supported:

> **b**   Use a general register other than zero. Some instructions treat the designation of register 0 specially, and do not behave as expected if the compiler chooses `r0`. For these instructions, the designation of `r0` does not mean that `r0` is used. Instead, it means that the literal value 0 is specified. See Example 8 for detailed usage information.

> **c**   Use the CTR register.

> **d**   Use a floating-point register.

> **f**   Use a floating-point register. See Example 7 for detailed usage information.

> **g**   Use a general register, memory, or immediate operand. In the

POWER® architecture, there are no instructions where a register, memory specifier, or immediate operand can be used interchangeably. However, this constraint is tolerated where it is possible to do so.

h   Use the CTR or LINK register.

i   Use an immediate integer or string literal operand.

l   Use the CTR register.

m   Use a memory operand supported by the machine. You can use this constraint for operands of the form `D(R)`, where `D` is a displacement and `R` is a register. See Example 9 for detailed usage information.

n   Use an immediate integer.

o   Use a memory operand that is offsetable. This means that the memory operand can be addressed by adding an integer to a base address. In the POWER architecture, memory operands are always offsetable, so the constraints `o` and `m` can be used interchangeably.

r   Use a general register. See Example 5 for detailed usage information.

s   Use a string literal operand.

v   Use a vector register.

0, 1, ...8, 9
A matching constraint. Allocate the same register in output as in the corresponding input.

I, J, K, L, M, N, O, P
Constant values. Fold the expression in the operand and substitute the value into the `%` specifier. These constraints specify a maximum value for the operand, as follows:

- I — signed 16-bit
- J — unsigned 16-bit shifted left 16 bits
- K — unsigned 16-bit constant
- L — signed 16-bit shifted left 16 bits
- M — unsigned constant greater than 31
- N — unsigned constant that is an exact power of 2
- O — zero
- P — signed whose negation is a signed 16-bit constant

C_expression

The *C_expression* is a C or C++ expression whose value is used as the operand for the `asm` instruction. Output operands must be modifiable lvalues. The *C_expression* must be consistent with the constraint specified on it. For example, if `i` is specified, the operand must be an integer constant number.

Related reference:

   See -fstrict-aliasing (-qalias=ansi) in the XL C/C++ Compiler Reference

Supported and unsupported constructs:

Supported constructs

The inline assembly statements support the following constructs:
- All the instruction statements listed in the Assembler Language Reference
- All extended instruction mnemonics
- Label definitions
- Branches to labels

Unsupported constructs

The inline assembly statements do not support the following constructs:
- Pseudo-operation statements, which are assembly statements that begin with a dot (.), such as `.function`
- Branches between different `asm` blocks

In addition, some constraints originating from the GNU compiler are not supported, but are tolerated where it is possible. For example, constraints S and T are treated as immediates, but the compiler issues a warning message stating that they are unsupported.

Restrictions on inline assembly statements:
The following restrictions are on the use of inline assembly statements:
- The assembler instructions must be self-contained within an `asm` statement. The `asm` statement can only be used to generate instructions. All connections to the rest of the program must be established through the output and input operand list.
- Referencing an external symbol directly without going through the operand list is not supported.
- Assembler instructions requiring a pair of registers are not specifiable by any constraints, and are therefore not supported. For example, you cannot use the %f constraint for a long double operand.
- The shared register file between the floating-point scalar and the vector registers are not modelled as shared in inline assembly statements. You must specify registers `f0-f31` and `v0-v31` in the clobbers list. There is no combined `x0-x63`.
- Operand replacements (such as `%0`, `%1`, and so on) can use an optional x before the number or symbolic name to indicate that a vsx register reference must be used. For example, a vector operand `%1` allocated to register v0 is replaced with 0 (for use in VMX instructions). The same operand used as `%x1` in the assembly text is replaced with 32 (for use in VSX instructions). Note that this restriction applies only for architectures that support VSX architecture extension.

Related reference:

See -fasm (-qasm) in the XL C/C++ Compiler Reference

Examples of inline assembly statements:
Example 1: The following example illustrates the usage of the `volatile` keyword.

```
#include <stdio.h>

inline bool acquireLock(int *lock){
  bool returnvalue = false;
  int lockval;
  asm volatile(
            /*--------a fence here-----*/
```

```
                     "  0: lwarx %0,0,%2   \n" // Loads the word and reserves
                                               // a memory location for the subsequent
                                               // stwcx. instruction.

                     "     cmpwi %0,0      \n" // Compares the lock value to 0.
                     "     bne- 1f         \n" // If it is 0, you can acquire the
                                               // lock. Otherwise, you did not get the
                                               // lock and must try again later.

                     "     ori %0,%0,1     \n" // Sets the lock to 1.
                     "     stwcx. %0,0,%2  \n" // Tries to conditionally store 1
                                               // into the lock word to acquire
                                               // the lock.

                     "     bne- 0b         \n" // Reservation was lost. Try again.

                     "     isync           \n" // Lock acquired. The isync instruction
                                               // implements an import barrier to
                                               // ensure that the instructions that
                                               // access the shared region guarded by
                                               // this lock are executed only after
                                               // they acquire the lock.

                     "     ori  %1,%1,1    \n" // Sets the return value for the
                                               // function acquireLock to true.

                     "  1:                 \n" // Did not get the lock.
                                               // Will return false.

                     /*------a fence here------*/

                     :     "+r"   (lockval),
                           "+r"   (returnvalue)
                     :     "r"    (lock)        // "lock" is the address of the lock in
                                                // memory.

                     :     "cr0"               // cr0 is clobbered by cmpwi and stwcx.
                     );

  return returnvalue;
}
int main()
{
  int myLock;
  if(acquireLock(&myLock)){
      printf("got it!\n");
  }else{
      printf("someone else got it\n");
  }
  return 0;
}
```

In this example, %0 refers to the first operand "+r"(lockval), %1 refers to the
second operand "+r"(returnvalue), and %2 refers to the third operand "r"(lock).

The assembly statement uses a lock to control access to the shared storage; no
instruction can access the shared storage before acquiring the lock.

The volatile keyword implies fences around the assembly instruction group, so
that no assembly instructions can be moved out of or around the assembly block.

Without the volatile keyword, the compiler can move the instructions around for
optimization. This might cause some instructions to access the shared storage
without acquiring the lock.

It is unnecessary to use the `memory` clobber in this assembly statement, because the instructions do not modify memory in an unexpected way. If you use the `memory` clobber, the program is still functionally correct. However, the `memory` clobber results in many unnecessary reloads, imposing a performance penalty.

Example 2: The following example illustrates the use of the symbolic names for input and output operands.

```
int a ;
int b = 1, c = 2, d = 3 ;
__asm("  addc %[result], %[first], %[second]"
     : [result]       "=r"       (a)
     : [first]        "r"        (b),
       [second]       "r"        (d)
     );
```

In this example, `%[result]` refers to the output operand variable a, `%[first]` refers to the input operand variable b, and `%[second]` refers to the input operand variable d.

Example 3: The following example shows a typical use of condition registers in the *clobbers*.

```
  asm ("    add. %0,%1,%2  \n"
       :    "=r"    (c)
       :    "r"     (a),
            "r"     (b)
       :    "cr0"
       );
```

In this example, apart from the registers listed in the *input* and *output* of the assembly statement, the `add.` instruction also affects the condition register field 0. Therefore, you must inform the compiler about this by adding `cr0` to the *clobbers*.

Example 4: The following example shows the usage of the `memory` clobber.

```
 asm volatile ("   dcbz 0, %0      \n"
               :    "=r"(b)
               :
               :    "memory"
               );
```

In this example, the instruction `dcbz` clears a cache block, and might have changed the variables in the memory location. There is no way for the compiler to know which variables have been changed. Therefore, the compiler assumes that all data might be aliased with the memory changed by that instruction.

As a result, everything that is needed must be reloaded from memory after the completion of the assembly statement. The `memory` clobber ensures program correctness at the expense of program performance, because the compiler might reload data that had nothing to do with the assembly statement.

Example 5: The following example shows the usage of the = modifier and the r constraint.

```
int a ;
int b = 100 ;
int c = 200 ;
asm("   add %0, %1, %2"
    :   "=r"      (a)
    :   "r"       (b),
        "r"       (c)
    );
```

The add instruction adds the contents of two general purpose registers. The %0, %1, and %2 operands are substituted by the C expressions in the output/input operand fields.

The output operand uses the = modifier to indicate that a modifiable operand is required; it uses the r constraint to indicate that a general purpose register is required. Likewise, the r constraint in the input operands indicates that general purpose registers are required. Within these restrictions, the compiler is free to choose any registers to substitute for %0, %1, and %2.

Note: If the compiler chooses r0 for the second operand, the add instruction uses the literal value 0 and yields an unexpected result. Thus, to prevent the compiler from choosing r0 for the second operand, you can use the b constraint to denote the second operand.

Example 6: The following example shows the usage of the + modifier and the K constraint.

```
asm ("    addi %0,%0,%2"
     :  "+r"   (a)
     :  "r"    (a),
        "K"    (15)
     );
```

This assembly statement adds operand %0 and operand %2, and writes the result to operand %0. The output operand uses the + modifier to indicate that operand %0 can be read and written by the instruction. The K constraint indicates that the value loaded to operand %2 must be an unsigned 16-bit constant value.

Example 7: The following example shows the usage of the % modifier and the f constraint.

```
asm("    fadd %0, %1, %2"
    :  "=f"   (c)
    :  "%f"   (a),
       "f"    (b)
    );
```

This assembly statement adds operands a and b, and writes the result to operand c. The % modifier indicates that operands a and b can be switched if the compiler can generate better code in doing so. Each operand has the f constraint, which indicates that a floating point register is required.

Example 8: The following example shows the usage of the b constraint.

```
char res[8]={'a','b','c','d','e','f','g','h'};
char a='y';
int  index=7;

asm ("   stbx %0,%1,%2       \n"      \
     :                                \
     :   "r"    (a),
         "b"    (index),
         "r"    (res)
     );
```

In this example, the b constraint instructs the compiler to choose a general register other than r0 for the input operand %1. The result string of this program is abcdefgy. However, if you use the r constraint and the compiler chooses r0 for %1,

this instruction produces an incorrect result string `ybcdefgh`. For instructions that treat the designation of `r0` specially, it is therefore important to denote the input operands with the `b` constraint.

Example 9: The following example shows the usage of the `m` constraint.

```
asm ("    stb %1,%0        \n"      \
    :   "=m"    (res)           \
    :   "r"     (a)
    );
```

In this example, the syntax of the instruction `stb` is `stb RS,D(RA)`, where `D` is a displacement and `R` is a register. `D+RA` forms an effective address, which is calculated from `D(RA)`. By using constraint **m**, you do not need to manually construct effective addresses by specifying the register and displacement separately.

You can use a single constraint `m` or `o` to refer to the two operands in the instruction, regardless of what the correct offset should be and whether it is an offset off the stack or off the TOC (Table of Contents). This allows the compiler to choose the right register (`r1` for an automatic variable, for instance) and apply the right displacement automatically.

The #include_next directive

The preprocessor directive `#include_next` behaves like the `#include` directive, except that it specifically excludes the directory of the including file from the paths to be searched for the named file. All search paths up to and including the directory of the including file are omitted from the list of paths to be searched for the included file. This allows you to include multiple versions of a file with the same name in different parts of an application; or to include one header file in another header file with the same name (without the header including itself recursively). Provided that the different file versions are stored in different directories, the directive ensures you can access each version of the file, without requiring that you use absolute paths to specify the file name.

#include_next directive syntax

```
►►—#—include_next—┬—"—┬──────────┬—file_name—"—┬───────────────────►◄
                  │    └─file_path─┘             │
                  └─<─┬──────────┬—file_name—>─┘
                      └─file_path─┘
```

The directive must only be used in header files, and the file specified by the *file_name* must be a header file. There is no distinction between the use of double quotation marks and angle brackets to enclose the file name.

As an example of how search paths are resolved with the `#include_next` directive, assume that there are two versions of the file `t.h`: the first one, which is included in the source file `t.c`, is located in the subdirectory `path1`; the second one, which is included in the first one, is located in the subdirectory `path2`. Both directories are specified as include file search paths when `t.c` is compiled.

```
/* t.c  */

#include "t.h"

int main()
{
printf(", ret_val);
}
```

```
/* t.h in path1 */

#include_next "t.h"

int ret_val = RET;

/* t.h in path2 */

#define RET 55;
```

The #include_next directive instructs the preprocessor to skip the path1 directory and start the search for the included file from the path2 directory. This directive allows you to use two different versions of t.h and it prevents t.h from being included recursively.

The __thread storage class specifier

The __thread storage class marks a static variable as having *thread-local* storage duration. This means that, in a multithreaded application, a unique instance of the variable is created for each thread that uses it, and destroyed when the thread terminates. The __thread storage class specifier can provide a convenient way of assuring thread-safety: declaring an object as per-thread allows multiple threads to access the object without the concern of race conditions, while avoiding the need for low-level programming of thread synchronization or significant program restructuring.

The tls_model attribute allows source-level control for the thread-local storage model used for a given variable. The tls_model attribute must specify one of local-exec, initial-exec, local-dynamic, or global-dynamic access method, which overrides the **-ftls-model (-qtls)** option for that variable. For example:

```
__thread int i __attribute__((tls_model("local-exec")));
```

The tls_model attribute allows the linker to check that the correct thread model has been used to build the application or shared library. The linker/loader behavior is as follows:

Table 11. Link time/runtime behavior for thread access models

Access method	Link-time diagnostic	Runtime diagnostic
local-exec	Fails if referenced symbol is imported.	Fails if module is not the main program. Fails if referenced symbol is imported (but the linker should have detected the error already).
initial-exec	None.	dlopen() fails if referenced symbol is not in the module loaded at execution time.
local-dynamic	Fails if referenced symbol is imported.	Fails if referenced symbol is imported (but the linker should have detected the error already).
global-dynamic	None.	None.

The specifier can be applied to variables with static storage duration. It cannot be applied to function-scoped or block-scoped automatic variables or non-static data members.

The thread specifier can be either preceded or followed by the `static` or `extern` specifier.

```
__thread int i;
extern __thread struct state s;
static __thread char *p;
```

> `C++` A thread variable must be initialized with a constant expression. `C++` <

Applying address operator (&) to a thread-local variable returns the runtime address of the current thread's instance of the variable. That thread can pass this address to any other thread; however, when the first thread terminates, any pointers to its thread-local variables become invalid.

Related reference:

See -ftls-model (-qtls) in the XL C/C++ Compiler Reference

The `typeof` keyword

The `typeof` and `__typeof__` keywords are supported as follows:

- > `C` The `__typeof__` keyword is recognized at all language levels. The `typeof` token is not a keyword at the **stdc89** and **stdc99** language levels. At all other language levels, `typeof` is treated as a keyword.

- > `C++` The `typeof` and `__typeof__` keywords are recognized by default.

Type attributes

Type attributes are language extensions provided to facilitate compilation of programs developed with the GNU C/C++ compilers. These language features allow you to use named attributes to specify special properties of data objects. Any variables that are declared as having that type will have the attribute applied to them.

A type attribute is specified with the keyword `__attribute__` followed by the attribute name and any additional arguments the attribute name requires. Although there are variations, the syntax of a type attribute is of the general form:

Type attribute syntax

```
►►─type_name─__attribute__─((─┬─►─┬─attribute name─────┬─┬─))────────►
                              │   └─__attribute name__─┘ │
                              └──────────,──────────────┘

►─┬──────────────────┬─{─member_definition_list─}─;────────────────►◄
  └─tag_identifier──┘
```
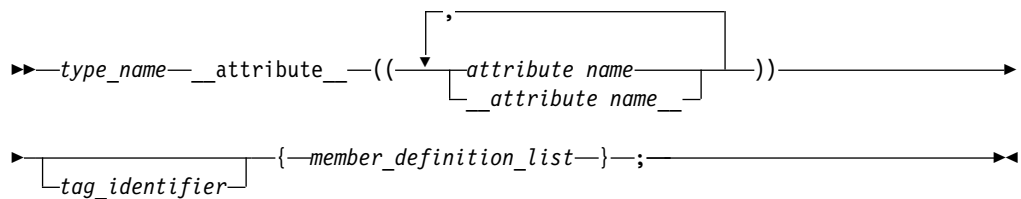
Type attribute syntax — typedef declarations

```
►►─typedef─type_declaration─type_name───────────────────────────────►
```
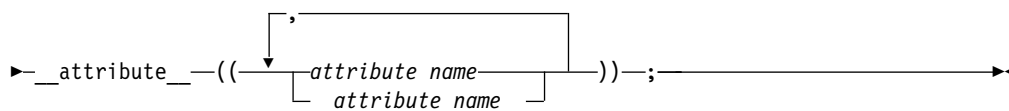
```
          ┌─────,─────────────────────┐
►─__attribute__─((─▼─┬─attribute name───┬──))─;──────────────────►◄
                     └─__attribute name__─┘
```

You can specify *attribute name* with or without leading and trailing double
underscore characters; however, using the double underscore characters reduces
the likelihood of name conflicts with macros of the same name. For unsupported
attribute names, the XL C/C++ compiler issues diagnostics and ignores the
attribute specification. Multiple attribute names can be specified in the same
attribute specification.

The following type attributes are supported:
- "The aligned type attribute"
- "The packed type attribute" on page 32
- "The may_alias type attribute" on page 31
- "The transparent_union type attribute" on page 49
- "The visibility type attribute" on page 53

Related reference:

"Variable attributes" on page 32

"Function attributes" on page 9

The aligned type attribute:
With the `aligned` type attribute, you can override the default alignment mode to
specify a minimum alignment value, expressed as a number of bytes, for a
structure, ► C++ classes C++ ◄ , union, enumeration, or other user-defined type
created in a `typedef` declaration. The `aligned` attribute is typically used to increase
the alignment of any variables declared of the type to which the attribute applies.

aligned type attribute syntax
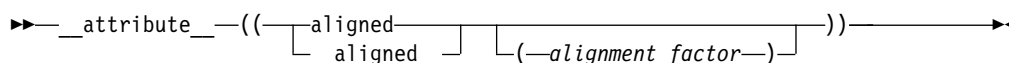
```
►►─__attribute__─((─┬─aligned────┬──┬─────────────────────────┬─))──────────►◄
                    └─__aligned__─┘  └─(─alignment_factor─)─┘
```

The *alignment_factor* is the number of bytes, specified as a constant expression that
evaluates to a positive power of 2. If you omit the alignment factor (and its
enclosing parentheses), the compiler automatically uses 16 bytes. You can specify a
value up to a maximum 268435456 bytes. If you specify an alignment factor greater
than the maximum, the compiler issues an error message and the compilation fails.

The alignment value that you specify is applied to all instances of the type. Also,
the alignment value applies to the variable as a whole; if the variable is an
aggregate, the alignment value applies to the aggregate as a whole, not to the
individual members of the aggregate.

Example

In all of the following examples, the `aligned` attribute is applied to the structure
type A. Because a is declared as a variable of type A, it also receives the alignment
specification, as any other instances declared of type A.

```
struct __attribute__((__aligned__(8))) A {};

struct __attribute__((__aligned__(8))) A {} a;

typedef struct __attribute__((__aligned__(8))) A {} a;
```

Related reference:

"The aligned variable attribute" on page 33

See Aligning data in the XL C/C++ Optimization and Programming Guide

The may_alias type attribute:
You can specify the may_alias type attribute for a type so that lvalues of the type can alias objects of any type, similar to a char type. Types with the may_alias attribute are not subject to type-based aliasing rules.

may_alias type attribute syntax

```
►►──__attribute__──((──┬──may_alias──────┬──))────────────────────────►◄
                       └──__may_alias__──┘
```

You can specify the may_alias type attribute in the following ways:

```
struct __attribute__((__may_alias__)) my_struct {} *ps;
typedef long __attribute__((__may_alias__)) t_long;
typedef struct __attribute__((__may_alias__)) my_struct {} t_my_struct;
```

Instead of specifying **-fnostrict-aliasing**, you can alternatively specify the may_alias type attribute for a type to violate the ANSI aliasing rules when compiling expressions that contain lvalues of that type. For example:

```
#define __attribute__(x)      // Invalidates all __attribute__ declarations
typedef float __attribute__((__may_alias__)) t_float;

int main (void){
  int i = 42;
  t_float *pa = (t_float *) &i;
  *pa = 0;
  if (i == 42)
    return 1;
  return 0;
}
```

If you compile this code with the **-fstrict-aliasing (-qalias=ansi)** option at a high optimization level, such as **-03**, the executable program returns 1. Because the lvalue *pa is of type float, according to the ANSI aliasing rules, the assignment to lvalue *pa cannot modify the value of i, which is of type int.

If you remove the #define __attribute__(x) statement and compile the code with the same options as before, the executable program returns 0. Because the type of *pa is float __attribute__((__may_alias__)), *pa can alias any other object of any type, and the assignment to lvalue *pa can modify the value of i to 0.

Compared to the **-fno-strict-aliasing (-qalias=noansi)** compiler option, the may_alias type attribute can result in less pessimistic aliasing assumptions by the compiler and thus lead to more optimization opportunities.

▶ C ◀ This attribute is supported at the extc89, extc99, extended, and extc1x language levels. ▶ C ◀

> **C++** This attribute is supported at the `extended`, `extended0x`, and `extended1y` language levels. **C++** ◄

Related reference:

📄 See -qalias in the XL C/C++ Compiler Reference

The packed type attribute:
The `packed` type attribute specifies that the minimum alignment should be used for the members of a structure, class, union, or enumeration type. For structure, class, or union types, the alignment is one byte for a member and one bit for a bit field member. For enumeration types, the alignment is the smallest size that will accomodate the range of values in the enumeration. All members of all instances of that type will use the minimum alignment.

packed type attribute syntax

```
►►──__attribute__─((──┬─packed───┬──))──────────────────────────►◄
                      └─__packed__─┘
```

Unlike the `aligned` type attribute, the `packed` type attribute is not allowed in a `typedef` declaration.

Related reference:

"The packed variable attribute" on page 35

📄 See Aligning data in the XL C/C++ Optimization and Programming Guide

Variable attributes
Variable attributes are language extensions provided to facilitate the compilation of programs developed with the GNU C/C++ compilers. These language features allow you to use named attributes to specify special properties of data objects. *Variable* attributes apply to the declarations of simple variables, aggregates, and member variables of aggregates.

A variable attribute is specified with the keyword __attribute__ followed by the attribute name and any additional arguments the attribute name requires. A variable __attribute__ specification is included in the declaration of a variable, and can be placed before or after the declarator. Although there are variations, the syntax generally takes either of the following forms:

Variable attribute syntax: post-declarator

```
                              ┌─,──────────────────┐
►►──declarator──__attribute__─((──▼─┬─attribute name───┬──))──────────►◄
                                    └─__attribute name__─┘
```

Variable attribute syntax: pre-declarator

```
                                  ┌─,──────────────────┐
►►──type specifier──__attribute__─((──▼─┬─attribute name───┬──))──────────►
                                        └─__attribute name__─┘
```
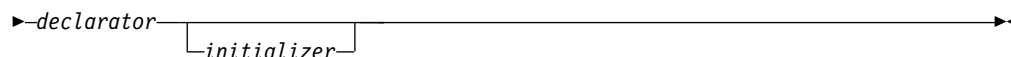
```
►──declarator──────────────────────────────────────────────────────►◄
          └─initializer─┘
```

You can specify *attribute name* with or without leading and trailing double
underscore characters; however, using the double underscore characters reduces
the likelihood of name conflicts with macros of the same name. For unsupported
attribute names, the XL C/C++ compiler issues diagnostics and ignores the
attribute specification. Multiple attribute names can be specified in the same
attribute specification.

In a comma-separated list of declarators on a single declaration line, if a variable
attribute appears before all the declarators, it applies to all declarators in the
declaration. If the attribute appears after a declarator, it only applies to the
immediately preceding declarator. For example:

```
struct A  {

 int b __attribute__((aligned));        /* typical placement of variable */
                                         /* attribute */
 int __attribute__((aligned)) c;        /* variable attribute can also be */
                                         /* placed here */

 int d, e, f __attribute__((aligned));  /* attribute applies to f only */

 int g __attribute__((aligned)), h, i;  /* attribute applies to g only */

 int __attribute__((aligned)) j, k, l;  /* attribute applies to j, k, and l */

 };
```

The following variable attributes are supported:
- "The aligned variable attribute"
- "The common and nocommon variable attributes" on page 34
- "The init_priority variable attribute" on page 52
- "The mode variable attribute" on page 35
- "The packed variable attribute" on page 35
- "The section variable attribute" on page 36
- "The tls_model attribute" on page 37
- "The weak variable attribute" on page 37
- "The visibility variable attribute" on page 37

Related reference:

"Type attributes" on page 29
"Function attributes" on page 9

The aligned variable attribute:
With the `aligned` variable attribute, you can override the default memory
alignment mode to specify a minimum memory alignment value, expressed as a
number of bytes, for any of the following types of variables:
- Non-aggregate variables
- Aggregate variables (such as a structures, classes, or unions)
- Selected member variables

The attribute is typically used to increase the alignment of the given variable.

aligned variable attribute syntax

```
►►──__attribute__──((──┬─aligned──────┬──┬──────────────────────┬──))───────────►◄
                       └─__aligned__──┘  └─(──alignment_factor──)─┘
```

The *alignment_factor* is the number of bytes, specified as a constant expression that evaluates to a positive power of 2. You can specify a value up to a maximum of 268435456. If you omit the alignment factor, and its enclosing parentheses, the compiler automatically uses 16 bytes. If you specify an alignment factor greater than the maximum, the compiler issues an error message and the compilation fails.

When you apply the aligned attribute to a member variable in a bit field structure, the attribute specification is applied to the bit field container. If the default alignment of the container is greater than the alignment factor, the default alignment is used.

Example

In the following example, the structures first_address and second_address are set to an alignment of 16 bytes:

```
struct address {
              int street_no;
              char *street_name;
              char *city;
              char *prov;
              char *postal_code;
           } first_address __attribute__((__aligned__(16))) ;

struct address second_address __attribute__((__aligned__(16))) ;
```

In the following example, only the members first_address.prov and first_address.postal_code are set to an alignment of 16 bytes:

```
struct address {
              int street_no;
              char *street_name;
              char *city;
              char *prov __attribute__((__aligned__(16))) ;
              char *postal_code __attribute__((__aligned__(16))) ;
           } first_address  ;
```

Related reference:

See Aligning data in the XL C/C++ Optimization and Programming Guide
"The aligned type attribute" on page 30

The common and nocommon variable attributes:
The variable attribute common allows you to specify that an uninitialized global variable or a global variable explicitly initialized to 0 should be allocated in the common section of the object file. The variable attribute nocommon specifies that an uninitialized global variable should be allocated in the data section of the object file. The variable is automatically initialized to zero.

nocommon and common variable attribute syntax

```
►►──__attribute__──((──┬──────────────┬──))─────────────────────────────────────►◄
                       ├─nocommon─────┤
                       ├─__nocommon__─┤
                       ├─common───────┤
                       └─__common__───┘
```

For example:
```
int i __attribute__((nocommon));     /* allocate i at .data */
int k __attribute__((common));       /* allocate k at .comm */
```

You can only apply the variable attributes to global scalar or aggregate variables. If you try to assign either attribute to a static or automatic variable or structure or union member, the attribute is ignored and a warning is issued.

Note that using `nocommon` to allocate uninitialized global variables in the data section can dramatically increase the size of the generated object. Also, specifying `nocommon` on a global variable that is simultaneously defined in different object files will cause an error at link time; such variables should be defined in one file and referred to in other files with an `extern` declaration.

The attributes take precedence over the -fcommon | -fno-common (-qcommon | -qnocommon) compiler option.

If multiple specifications of the attribute appear in the same attribute statement, the last one specified will take effect. For example:
```
int i __attribute__((common, nocommon));          /* allocate i at .data */
int k __attribute__((common, nocommon, common));  /* allocate k at .comm */
```

If both the `common` or `nocommon` attribute and the `section` attribute are applied to the same variable, the `section` attribute takes precedence.
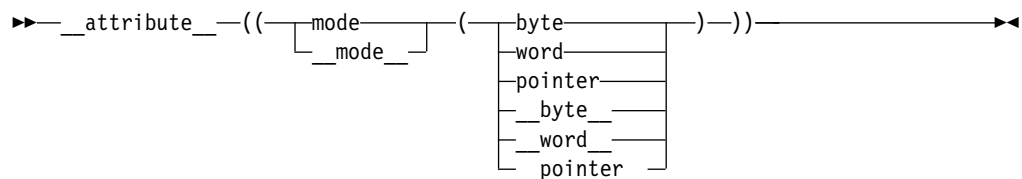
Related reference:

📄 See -qcommon in the XL C/C++ Compiler Reference

The mode variable attribute:
The variable attribute `mode` allows you to override the type specifier in a variable declaration, to specify the size of a particular integral type.

mode variable attribute syntax

```
►►──__attribute__──((──┬─mode───┬──(──┬─byte──────┬──)──))──────────►◄
                       └─__mode__┘     ├─word──────┤
                                       ├─pointer───┤
                                       ├─__byte__──┤
                                       ├─__word__──┤
                                       └─__pointer__┘
```

The valid argument for the mode is any of the of the following type specifiers that indicates a specific width:
- `byte` means a 1-byte integer type
- `word` means a 4-byte integer type
- `pointer` means an 8-byte integer type

The packed variable attribute:
The variable attribute `packed` allows you to override the default alignment mode, to reduce the alignment for all members of an aggregate, or selected members of an aggregate to the smallest possible alignment: one byte for a member and one bit for a bit field member.

packed variable attribute syntax

```
►►─┬─__attribute__─((─┬─packed───┬─))──────────────────────────►◄
                       └─__packed__─┘
```

Related reference:

📄 See Aligning data in the XL C/C++ Optimization and Programming Guide

The section variable attribute:
The `section` variable attribute specifies the section in the object file in which the compiler should place its generated code. The language feature provides the ability to control the section in which a variable should appear.

section variable attribute syntax

```
►►──declarator──────────────────────────────────────────────────►

►──__attribute__─((─┬─section───┬─(─"─section_name─"─)─))──────────►◄
                     └─__section__─┘
```

The *section_name* specifies a named section as a string literal, maximum length of 16 characters, not counting spaces. Spaces in the string are ignored.

The section variable attribute can be applied to a declaration or definition of the following types of variables:
- initialized or static global or namespace variables
- static local variables
- ► **C++** uninitialized global or namespace variables
- ► **C++** static structure or class member variables

A section attribute applied to a local variable with automatic storage duration is ignored with a warning because such variables are stored on the stack.

► **C** A section attribute applied to a structure member is ignored with a warning. A section attribute applied to an uninitialized global variable is ignored without a warning; the symbols for uninitialized global variables are always placed in the common section. **C** ◄

When multiple section attributes are applied to a variable declaration, the last specification prevails. The section indicated in the prevailing variable declaration should match that of the variable definition because a variable definition cannot be overwritten. Each defined variable can reside in only one section.

The `section` attribute overrides the **-fcommon | -fno-common (-qcommon | -qnocommon)** compiler option and the `common|nocommon` attribute. That is, if both attributes are specified for the same variable, the `section` attribute takes priority.

A named section can be used for multiple variables, but not for both variables and functions in the same compilation unit.

Related reference:

"The section function attribute" on page 16

"The common and nocommon variable attributes" on page 34

📄 See -qcommon in the XL C/C++ Compiler Reference

The tls_model attribute:
The `tls_model` attribute allows source-level control for the thread-local storage model used for a given variable. The `tls_model` attribute must specify one of `local-exec`, `initial-exec`, `local-dynamic`, or `global-dynamic` access method, which overrides the **-ftls-model (-qtls)** option for that variable. For example:

```
__thread int i __attribute__((tls_model("local-exec")));
```

The `tls_model` attribute allows the linker to check that the correct thread model has been used to build the application or shared library. The linker/loader behavior is as follows:

Table 12. Link time/runtime behavior for thread access models

Access method	Link-time diagnostic	Runtime diagnostic
`local-exec`	Fails if referenced symbol is imported.	Fails if module is not the main program. Fails if referenced symbol is imported (but the linker should have detected the error already).
`initial-exec`	None.	`dlopen()` fails if referenced symbol is not in the module loaded at execution time.
`local-dynamic`	Fails if referenced symbol is imported.	Fails if referenced symbol is imported (but the linker should have detected the error already).
`global-dynamic`	None.	None.

The weak variable attribute:
The `weak` variable attribute causes the symbol resulting from the variable declaration to appear in the object file as a weak symbol, rather than a global one. The language feature provides the programmer writing library functions with a way to allow variable definitions in user code to override the library declaration without causing duplicate name errors.

weak variable attribute syntax

```
►►──__attribute__──((──┬─weak────┬──))──────────────────────►◄
                       └─__weak__─┘
```
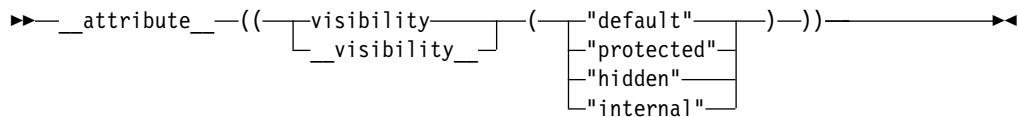
Related reference:
"weak" on page 17

The visibility variable attribute:
The visibility variable attribute describes whether and how a variable defined in one module can be referenced or used in other modules. The visibility attribute affects only variables with external linkage. By using this feature, you can make a shared library smaller and decrease the possibility of symbol collision. For details, see Using visibility attributes in the XL C/C++ Optimization and Programming Guide.

visibility variable attribute syntax

```
►►──__attribute__──((──┬─visibility────┬──(──┬─"default"──┬──)──))──────────────►◄
                       └─__visibility__─┘     ├─"protected"─┤
                                              ├─"hidden"────┤
                                              └─"internal"──┘
```

Example

In the following example, the visibility attribute of variable a is protected, and that of variable b is hidden:

```
struct str{
    int var;
};
int a __attribute__((visibility("protected")));
struct str __attribute__((visibility("hidden"))) b;
```

Related reference:

visibility

"The visibility type attribute" on page 53

"The visibility namespace attribute" on page 52

See Using visibility attributes in the XL C/C++ Optimization and Programming Guide

See -fvisibility (-qvisibility) in the XL C/C++ Compiler Reference

See -shared (-qmkshrobj) in the XL C/C++ Compiler Reference

See #pragma GCC visibility push (visibility), #pragma GCC visibility pop in the XL C/C++ Compiler Reference

Variadic macros

More complex than object-like macros, a function-like macro definition declares the names of formal parameters within parentheses, separated by commas. An empty formal parameter list is legal: such a macro can be used to simulate a function that takes no arguments. C99 adds support for function-like macros with a variable number of arguments. XL C++ supports function-like macros with a variable number of arguments, as a language extension for compatibility with C and as part of C++11.

> IBM

Variadic macro extensions

Variadic macro extensions refer to two extensions to C99 and C++03 related to macros with variable number of arguments. One extension is a mechanism for renaming the variable argument identifier from __VA_ARGS__ to a user-defined identifier. The other extension provides a way to remove the dangling comma in a variadic macro when no variable arguments are specified. Both extensions have been implemented to facilitate porting programs developed with GNU C and C++.

The following examples demonstrate the use of an identifier in place of __VA_ARGS__. The first definition of the macro debug exemplifies the usual usage of __VA_ARGS__. The second definition shows the use of the identifier args in place of __VA_ARGS__.

```
#define debug1(format, ...)  printf(format, ## __VA_ARGS__)
#define debug2(format, args ...)  printf(format, ## args)
```

Invocation	Result of macro expansion
debug1("Hello %s/n", "World");	printf("Hello %s/n", "World");
debug2("Hello %s/n", "World");	printf("Hello %s/n", "World");

The preprocessor removes the trailing comma if the variable arguments to a
function macro are omitted and the comma followed by ## precedes the variable
argument identifier in the function macro definition.

> IBM <

Extensions for vector processing support

The vector extensions are only accepted when all of the following conditions are
met:

- The **-mcpu** option is set to a target architecture that supports vector processing
  instructions. For example, an architecture that supports the VSX instruction set
  extensions, such as POWER8®, requires **-mcpu=pwr8**.
- The **-qaltivec** option is in effect.

For more information on these options, see the *XL C/C++ Compiler Reference*.

*Table 13. IBM XL C and C++ extensions to support the AltiVec Application Programming
Interface specification*

Language feature	Discussed in:
Vector programming language extensions	"Vector types" on page 46, "Vector literals" on page 42

The following features are IBM extensions to the AltiVec Application Programming
Interface specification:

*Table 14. IBM XL C and C++ extensions to the AltiVec Application Programming Interface
specification*

Language extension	Discussed in:
__pixel, pixel, __vector, and vector keywords	N/A
Indirection operator * applied to vector types	N/A
Initialization of vectors	"Initialization of vectors"
Pointer arithmetic on vector types	"Pointer arithmetic" on page 40
Unary and binary operators	"Unary and binary operators" on page 41
Vector types as arguments to vec_step	"The vec_step operator" on page 40
Vector type casts	"Vector type casts" on page 46
Vector subscripting operator [ ]	"Vector subscripting operator [ ]" on page 45

Initialization of vectors

A vector type is initialized by a vector literal or any expression having the same
vector type. For example:

```
vector unsigned int v = (vector unsigned int)(10);
```

The AltiVec specification allows a vector type to be initialized by an initializer list. This feature is an extension for compatibility with GNU C.

Unlike vector literals, the values in the initializer list do not have to be constant expressions except in contexts where a constant value is required; ▶ C ◀ the initialization of a global vector variable is one such context. ■ C ◀ Thus, the following code is legal:

```
int i=1;
int function() { return 2; }
int main()
{
   vector unsigned int v1 = {i, function()};
   return 0;
}
```

Pointer arithmetic

Pointer arithmetic is defined for pointer to vector types. Given:

```
vector unsigned int *v;
```

the expression v + 1 represents a pointer to the vector following v.

The vec_step operator

The vec_step operator takes a vector type operand and returns an integer value representing the amount by which a pointer to a vector element should be incremented in order to move by 16 bytes (the size of a vector). The following table provides a summary of values by data type.

Table 15. Increment value for vec_step by data type

vec_step expresssion	Value
vec_step(vector unsigned char) vec_step(vector signed char) vec_step(vector bool char)	16
vec_step(vector unsigned short) vec_step(vector signed short) vec_step(vector bool short)	8
vec_step(vector unsigned int) vec_step(vector signed int) vec_step(vector bool int)	4
vec_step(vector unsigned long long) vec_step(vector signed long long) vec_step(vector bool long long)	2
vec_step(vector pixel)	8
vec_step(vector float)	4
vec_step(vector double)	2

For complete information about the `vec_step` operator, see the *AltiVec Technology Programming Interface Manual*, available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

Unary and binary operators

Vector data types can use some of the unary operators and binary operators that are used with primitive data types. Note that all operators require compatible types as operands unless otherwise stated. These operators are not supported at global scope or for objects with static duration, and there is no constant folding.

Unary operators

For unary operators, each element in the vector has the operation applied to it.

Table 16. Unary operators

Operator	Integer vector types	Vector double	Bool vector types
++	Yes	Yes	No
−−	Yes	Yes	No
+	Yes	Yes	No
−	Yes (except unsigned vectors)	Yes	No
~	Yes	No	Yes
*[1]	Yes	Yes	Yes

Note:

1. Although the preferred alignment for vector data types is at a multiple of 16 bytes, pointers can point to vector objects at any alignment.

Binary operators

For binary operators, each element has the operation applied to it with the same position element in the second operand. Binary operators also include relational operators. For relational operators, each element has the operation applied to it with the same position element in the second operand and the results have the **AND** operator applied to them to get a final result of a single value.

Table 17. Binary operators

Operator	Integer vector types	Vector double	Bool vector types
=	Yes	Yes	Yes
+	Yes	Yes	No
−	Yes	Yes	No
*	Yes	Yes	No
/	Yes	Yes	No
%	Yes	No	No
&	Yes	No	Yes
|	Yes	No	Yes
^	Yes	No	Yes
<<	Yes	No	Yes
>>	Yes	No	Yes

Table 17. Binary operators (continued)

Operator	Integer vector types	Vector double	Bool vector types
[]¹	Yes	Yes	Yes
==	Yes	Yes	Yes
!=	Yes	Yes	Yes
<	Yes	Yes	No
>	Yes	Yes	No
<=	Yes	Yes	No
>=	Yes	Yes	No

Note:

1. The [] operator returns the vector element at the position specified. If the position specified is outside of the valid range, the behavior is undefined.

For the following code:

```
vector unsigned int a = {1,2,3,4};
vector unsigned int b = {2,4,6,8};
vector unsigned int c = a + b;
int e = b > a;
int f = a[2];
vector unsigned int d = ++a;
```

c would have the value (3,6,9,12), d would have the value (2,3,4,5), e would have a non-zero value, and f would have the value 3.

Vector literals

A vector literal is a constant expression for which the value is interpreted as a vector type. The data type of a vector literal is represented by a parenthesized vector type, and its value is a set of constant expressions that represent the vector elements and are enclosed in parentheses or braces. When all vector elements have the same value, the value of the literal can be represented by a single constant expression. You can initialize vector types with vector literals.

Vector literal syntax

```
►►──(─vector_type─)──┬─(─literal_list─)─┬──────────────────────►◄
                     └─{─literal_list─}─┘
```

literal_list:

```
      ┌─,───────────────────┐
├──────▼─constant_expression─┴──────────────────────────────┤
```

The *vector_type* is a supported vector type; see "Vector types" on page 46 for a list of these.

The *literal_list* can be either of the following expressions:
- A single expression.

  If the single expression is enclosed with parentheses, all elements of the vector are initialized to the specified value. If the single expression is enclosed with

braces, the first element of the vector is initialized to the specified value, and the remaining elements of the vector are initialized to 0.

- A comma-separated list of expressions. Each element of the vector is initialized to the respectively specified value.

  The number of constant expressions is determined by the type of the vector and whether it is enclosed with braces or parentheses.

  If the comma-separated list of expressions is enclosed with braces, the number of constant expressions can be equal to or less than the number of elements in the vector. If the number of constant expressions is less than the number of elements in the vector, the values of the unspecified elements are 0.

  If the comma-separated list of expressions is enclosed with parentheses, the number of constant expressions must match the number of elements in the vector as follows:

  2   For `vector long long`, `vector bool long long`, and `vector double` types.

  4   For `vector int` and `vector float` types.

  8   For `vector short` and `vector pixel` types.

  16   For `vector char` types.

The following table shows the supported vector literals and how the compiler interprets them to determine their values.

Table 18. Vector literals

Syntax	Interpreted by the compiler as
(vector unsigned char)(*unsigned int*) (vector unsigned char){*unsigned int*}	A set of 16 unsigned 8-bit quantities that all have the value of the single integer.
(vector unsigned char)(*unsigned int, ...*) (vector unsigned char){*unsigned int, ...*}	A set of 16 unsigned 8-bit quantities with the value specified by each of the 16 integers.
(vector signed char)(*int*) (vector signed char){*int*}	A set of 16 signed 8-bit quantities that all have the value of the single integer.
(vector signed char)(*int, ...*) (vector signed char){*int, ...*}	A set of 16 signed 8-bit quantities with the value specified by each of the 16 integers.
(vector bool char)(*unsigned int*) (vector bool char){*unsigned int*}	A set of 16 unsigned 8-bit quantities that all have the value of the single integer.
(vector bool char)(*unsigned int, ...*) (vector bool char){*unsigned int, ...*}	A set of 16 unsigned 8-bit quantities with a value specified by each of 16 integers.
(vector unsigned short)(*unsigned int*) (vector unsigned short){*unsigned int*}	A set of 8 unsigned 16-bit quantities that all have the value of the single integer.
(vector unsigned short)(*unsigned int, ...*) (vector unsigned short){*unsigned int, ...*}	A set of 8 unsigned 16-bit quantities with a value specified by each of the 8 integers.
(vector signed short)(*int*) (vector signed short){*int*}	A set of 8 signed 16-bit quantities that all have the value of the single integer.

Table 18. Vector literals (continued)

Syntax	Interpreted by the compiler as
(vector signed short)(*int*, ...) (vector signed short){*int*, ...}	A set of 8 signed 16-bit quantities with a value specified by each of the 8 integers.
(vector bool short)(*unsigned int*) (vector bool short){*unsigned int*}	A set of 8 unsigned 16-bit quantities that all have the value of the single integer.
(vector bool short)(*unsigned int*, ...) (vector bool short){*unsigned int*, ...}	A set of 8 unsigned 16-bit quantities with a value specified by each of the 8 integers.
(vector unsigned int)(*unsigned int*) (vector unsigned int){*unsigned int*}	A set of 4 unsigned 32-bit quantities that all have the value of the single integer.
(vector unsigned int)(*unsigned int*, ...) (vector unsigned int){*unsigned int*, ...}	A set of 4 unsigned 32-bit quantities with a value specified by each of the 4 integers.
(vector signed int)(*int*) (vector signed int){*int*}	A set of 4 signed 32-bit quantities that all have the value of the single integer.
(vector signed int)(*int*, ...) (vector signed int){*int*, ...}	A set of 4 signed 32-bit quantities with a value specified by each of the 4 integers.
(vector bool int)(*unsigned int*) (vector bool int){*unsigned int*}	A set of 4 unsigned 32-bit quantities that all have the value of the single integer.
(vector bool int)(*unsigned int*, ...) (vector bool int){*unsigned int*, ...}	A set of 4 unsigned 32-bit quantities with a value specified by each of the 4 integers.
(vector unsigned long long)(*unsigned long long*) (vector unsigned long long){*unsigned long long*}	A set of 2 unsigned 64-bit quantities that both have the value of the single long long.
(vector unsigned long long)(*unsigned long long*, ...) (vector unsigned long long){*unsigned long long*, ...}	A set of 2 unsigned 64-bit quantities specified with a value by each of the 2 unsigned long longs.
(vector signed long long)(*signed long long*) (vector signed long long){*signed long long*}	A set of 2 signed 64-bit quantities that both have the value of the single long long.
(vector signed long long)(*signed long long*, ...) (vector signed long long){*signed long long*, ...}	A set of 2 signed 64-bit quantities with a value specified by each of the 2 long longs.
(vector bool long long)(*unsigned long long*) (vector bool long long){*unsigned long long*}	A set of 2 boolean 64-bit quantities with a value specified by the single unsigned long long.
(vector bool long long)(*unsigned long long*, ...) (vector bool long long){*unsigned long long*, ...}	A set of 2 boolean 64-bit quantities with a value specified by each of the 2 unsigned long longs.
(vector float)(*float*) (vector float){*float*}	A set of 4 32-bit single-precision floating-point quantities that all have the value of the single float.

Table 18. Vector literals (continued)

Syntax	Interpreted by the compiler as
(vector float)(*float*, ...) (vector float){*float*, ...}	A set of 4 32-bit single-precision floating-point quantities with a value specified by each of the 4 floats.
(vector double)(*double*) (vector double){*double*}	A set of 2 64-bit double-precision floating-point quantities that both have the value of the single double.
(vector double)(*double, double*) (vector double){*double, double*}	A set of 2 64-bit double-precision floating-point quantities with a value specified by each of the 2 doubles.
(vector pixel)(*unsigned int*) (vector pixel){*unsigned int*}	A set of 8 unsigned 16-bit quantities that all have the value of the single integer.
(vector pixel)(*unsigned int*, ...) (vector pixel){*unsigned int*, ...}	A set of 8 unsigned 16-bit quantities with a value specified by each of the 8 integers.

Note: The value of an element in a vector bool is FALSE if each bit of the element is set to 0 and TRUE if each bit of the element is set to 1.

For example, for an unsigned integer vector type, the literal could be either of the following:

```
(vector unsigned int)(10)   /* initializes all four elements to a value of 10  */
(vector unsigned int)(14, 82, 73, 700)   /* initializes the first element
                                    to 14, the second element to 82,
                                    the third element to 73, and the
                                    fourth element to 700   */
```

You can cast vector literals with the cast operator (). Enclosing the vector literal to be cast in parentheses can improve the readability of the code. For example, you can use the following code to cast a `vector signed int` literal to a `vector unsigned char` literal:

```
(vector unsigned char)((vector signed int)(-1, -1, 0, 0))
```

Related reference:

"Vector types" on page 46

"Initialization of vectors" on page 39

Vector subscripting operator [ ]

Access to individual elements of a vector data type is provided through the use of square brackets, similar to how array elements are accessed. The vector data type is followed by a set of square brackets containing the position of the element. The position of the first element is 0. The type of the result is the type of the elements contained in the vector type.

Example:

```
vector unsigned int v1 = {1,2,3,4};
unsigned int u1, u2, u3, u4;
u1 = v1[0];        // u1=1
u2 = v1[1];        // u2=2
u3 = v1[2];        // u3=3
u4 = v1[3];        // u4=4
```

Note: You can also access and manipulate individual elements of vectors with the following intrinsic functions:

- **vec_extract**
- **vec_insert**
- **vec_promote**
- **vec_splats**

Vector type casts

Vector types can be cast to other vector types. The cast does not perform a conversion: it preserves the 128-bit pattern, but not necessarily the value. A cast between a vector type and a scalar type is not allowed.

Vector pointers and pointers to non-vector types can be cast back and forth to each other. When a pointer to a non-vector type is cast to a vector pointer, the address should be 16-byte aligned. The referenced object of the pointer to a non-vector type can be aligned on a 16-byte boundary by using __attribute__((aligned(16))).

Vector types

XL C/C++ supports vector processing technologies through language extensions. XL C/C++ implements and extends the AltiVec Programming Interface specification. In the extended syntax, type qualifiers and storage class specifiers can precede the keyword vector (or its alternative spelling, __vector) in a declaration.

Most of the legal forms of the syntax are captured in the following diagram. Some variations have been omitted from the diagram for the sake of clarity: type qualifiers such as const and storage class specifiers such as static can appear in any order within the declaration, as long as neither immediately follows the keyword vector (or __vector).

Vector declaration syntax
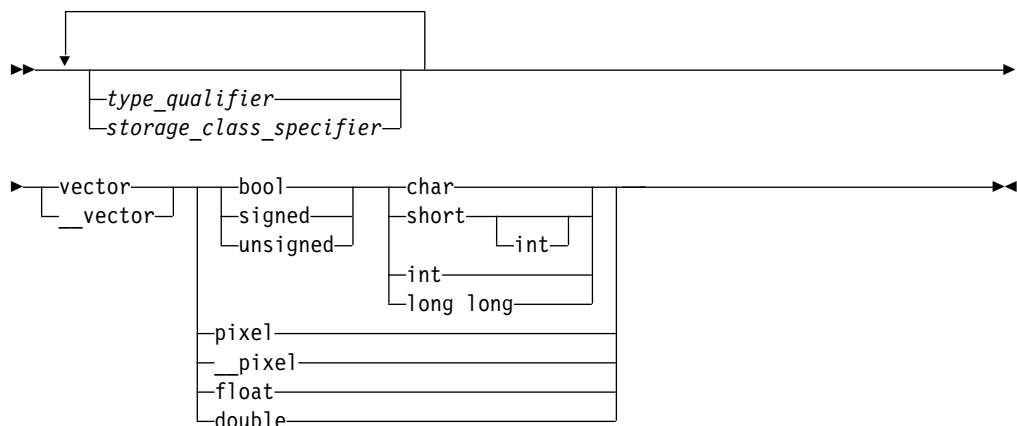
Notes:

- The keyword vector is recognized in a declaration context only when used as a type specifier and when vector support is enabled. The keywords pixel, __pixel and bool are recognized as valid type specifiers only when preceded by the keyword vector or __vector.

- Duplicate type specifiers are ignored in a vector declaration context.

The following table lists the supported vector data types, the size and possible values for each type.

Table 19. Vector data types

Type	Interpretation of content	Range of values
vector unsigned char	16 unsigned char	0..255
vector signed char	16 signed char	-128..127
vector bool char	16 unsigned char	0, 255
vector unsigned short vector unsigned short int	8 unsigned short	0..65535
vector signed short vector signed short int	8 signed short	-32768..32767
vector bool short vector bool short int	8 unsigned short	0, 65535
vector unsigned int	4 unsigned int	$0..2^{32}-1$
vector signed int	4 signed int	$-2^{31}..2^{31}-1$
vector bool int	4 unsigned int	$0, 2^{32}-1$
vector unsigned long long	2 unsigned long long	$0..2^{64}-1$
vector bool long long		$0, 2^{64}-1$
vector signed long long	2 signed long long	$-2^{63}..2^{63}-1$
vector float	4 float	IEEE-754 single (32 bit) precision floating-point values
vector double	2 double	IEEE-754 double (64 bit) precision floating-point values
vector pixel	8 unsigned short	1/5/5/5 pixel

All vector types are aligned on a 16-byte boundary. An aggregate that contains one or more vector types is aligned on a 16-byte boundary, and padded, if necessary, so that each member of vector type is also 16-byte aligned.

Related reference:

Vector literals

"Initialization of vectors" on page 39

"The aligned variable attribute" on page 33

IBM extension features for C only

This section describes IBM extension features for the C language only in the following categories:
- "Extensions for GNU C compatibility" on page 48
- "Extensions for vector processing support" on page 50

Extensions for GNU C compatibility

The following features are enabled by default when you compile with any of the following commands:

- the **xlc** invocation command
- the **-qlanglvl=extc99 | extc89 | extended** options

Table 20. Default IBM XL C extensions for GNU C compatibility

Language feature	Discussed in:
Cast to a union type	"Cast to union type"
The `transparent_union` type attribute	"The transparent_union type attribute" on page 49

Cast to union type

Casting to a union type is the ability to cast a union member to the same type as the union to which it belongs. Such a cast does not produce an lvalue. The feature is supported as an extension to C99, implemented to facilitate porting programs developed with GNU C.

Only a type that explicitly exists as a member of a union type can be cast to that union type. The cast can use either the tag of the union type or a union type name declared in a `typedef` expression. The type specified must be a complete union type. An anonymous union type can be used in a cast to a union type, provided that it has a tag or type name. A bit field can be cast to a union type, provided that the union contains a bit field member of the same type, but not necessarily of the same length. The following code shows an example of a simple cast to union:

```
#include <stdio.h>

union f {
   char t;
   short u;
   int v;
   long w;
   long long x;
   float y;
   double z;
};

int main() {
  union f u;
  char a = 1;
  u = (union f)a;
  printf("u = %i\n", u.t);
}
```

The output of this example is:

```
u = 1
```

Casting to a nested union is also allowed. In the following example, the `double` type dd can be cast to the nested union u2_t.

```
int main() {
   union u_t {
      char a;
      short b;
      int c;
      union u2_t {
         double d;
      }u2;
```

```
  };
  union u_t U;
  double dd = 1.234;
  U.u2 = (union u2_t) dd;        // Valid.
  printf("U.u2 is %f\n", U.u2);
}
```

The output of this example is:

```
U.u2 is 1.234
```

A union cast is also valid as a function argument, part of a constant expression for initialization of a static or non-static data object, and in a compound literal statement. The following example shows a cast to union used as part of an expression for initializing a static object:

```
struct S{
  int a;
}s;

union U{
  struct S *s;
};

struct T{
  union U u;
};

static struct T t[] = { {(union U)&s} };
```

Related reference:

"The transparent_union type attribute"

The transparent_union type attribute

The `transparent_union` attribute applied to a union definition or a union `typedef` definition indicates the union can be used as a *transparent union*. The union must be a complete union type.

Whenever a transparent union is the type of a function parameter and that function is called, the transparent union can accept an argument of any type that matches that of one of its members without an explicit cast. Arguments to this function parameter are passed to the transparent union, using the calling convention of the first member of the union type. Because of this, all members of the union must have the same machine representation. Transparent unions are useful in library functions that use multiple interfaces to resolve issues of compatibility.

transparent_union type attribute syntax

```
►►──__attribute__──((──┬─transparent_union───┬──))──────────────────────►◄
                       └─__transparent_union__─┘
```

When the `transparent_union` type attribute is applied to the outer union of a nested union, the size of the inner union (that is, its largest member) is used to determine if it has the same machine representation as the other members of the outer union. See the following example:

```
union u_t{
    union u2_t{
        char a;
        short b;
        char c;
```

```
        char d;
    }u;
    int a1;
}__attribute__((__transparent_union__));
```

The attribute is ignored because the first member of union u_t, which is itself a union, has a machine representation of 2 bytes, whereas the other member of union u_t is of type int, which has a machine representation of 4 bytes.

The same rationale applies to members of a union that are structures. When a member of a union to which type attribute transparent_union has been applied is a struct, the machine representation of the entire struct is considered, rather than members.

All members of the union must have the same machine representation as the first member of the union. This means that all members must be representable by the same amount of memory as the first member of the union. The machine representation of the first member represents the maximum memory size for any remaining union members. For instance, if the first member of a union to which type attribute transparent_union has been applied is of type int, then all following members must be representable by at most 4 bytes. Members that are representable by 1, 2, or 4 bytes are considered valid for this transparent union.

Floating-point types (float, double, float _Complex, or double _Complex) types or vector types can be members of a transparent union, but they cannot be the first member. The restriction that all members of the transparent union have the same machine representation as the first member still applies.

Extensions for vector processing support

The following feature is an IBM extension to the AltiVec Application Programming Interface specification:

Table 21. IBM XL C extension to the AltiVec Application Programming Interface specification

Language extension	Discussed in:
bool keyword	N/A

IBM extension features for C++ only

This section describes IBM extension features for the C++ language only in the following categories:
- "Extensions for C99 compatibility"
- "Extensions for C11 compatibility" on page 51
- "Extensions for GNU C++ compatibility" on page 51

Extensions for C99 compatibility

IBM XL C++ adds support for the following C99 language features. All of these features are enabled by default.
- _Complex keyword
- __func__ predefined identifier
- Complex data type
- Compound literals
- C standard pragmas

- Duplicate type qualifiers
- Flexible array members at the end of a structure or union
- Hexadecimal floating-point literals
- The restrict type qualifier
- Universal character names
- Variable length arrays

Note: The _Imaginary keyword is reserved for possible future use. For complex number functionality, use _Complex.

Extensions for C11 compatibility

IBM XL C++ adds support for some C11 language features.

The following features are enabled by default:
- _Noreturn function specifier
- _Thread_local
- Atomics
- Composite types for variable length arrays
- Conversions between pointers and floating types
- Generic selection
- Static assertions
- Temporary lifetime extensions
- typedef redeclarations
- Unicode and UTF-8 literals

The complex type initializations feature is enabled when you specify the **-qlanglvl=extended** or **-qlanglvl=extended0x** option.

Extensions for GNU C++ compatibility

The following GNU C++ language extensions are enabled by default.

Table 22. IBM XL C++ language extensions for compatibility with GNU C++

Language feature	Discussed in:
__decltype keyword	N/A
init_priority variable attribute	"The init_priority variable attribute" on page 52

The following GNU C++ language extensions are enabled by default. They can also be enabled or disabled by specific compiler options, listed in the below table:

Table 23. IBM XL C++ language extensions for compatibility with GNU C++, with individual option controls

Language feature	Discussed in:	Individual option control
Visibility namespace attribute	"The visibility namespace attribute" on page 52	**-fvisibility**
Visibility type attribute	"The visibility type attribute" on page 53	**-fvisibility**

Table 23. IBM XL C++ language extensions for compatibility with GNU C++, with individual option controls (continued)

Language feature	Discussed in:	Individual option control
Note: You can use the **-fvisibility** option to specify visibility attributes for types and namespaces if they do not get visibility attributes from pragma directives, explicitly specified attributes, or propagation rules. This option cannot be used to disable visibility attributes for types or namespaces.		

The init_priority variable attribute

The variable attribute init_priority is an extension to C++ that allows you to control the initialization order of static objects defined in namespace scope across multiple compilation units.

init_priority variable attribute syntax

The *relative_priority* is a constant integral expression between 101 and 65535, inclusive. A lower number indicates a higher priority.

The visibility namespace attribute

The visibility namespace attribute is a language extension that allows you to control whether and how the entities within a namespace defined in one module can be referenced or used in other modules. By using this feature, you can make a shared library smaller and decrease the possibility of symbol collision. For details, see Using visibility attributes in the XL C/C++ Optimization and Programming Guide.

visibility namespace attribute syntax

You can specify the attribute name visibility with or without leading and trailing double underscore characters; however, using the double underscore characters reduces the likelihood of name conflicts with macros of the same name.

Example

In the following example, function fun() is defined in namespace A , and the visibility attribute of fun() is default:

```
namespace A __attribute__((visibility("default"))) {
   void fun(){}
}
```

Related reference:

"The visibility variable attribute" on page 37

visibility

"The visibility type attribute" on page 53

See Using visibility attributes in the XL C/C++ Optimization and Programming Guide

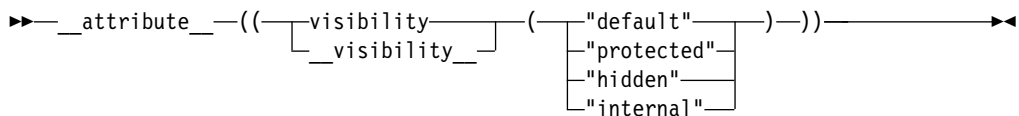See -fvisibility (-qvisibility) in the XL C/C++ Compiler Reference

See -shared (-qmkshrobj) in the XL C/C++ Compiler Reference

See #pragma GCC visibility push (visibility), #pragma GCC visibility pop in the XL C/C++ Compiler Reference

The visibility type attribute

With the visibility type attributes, you can control whether and how a structure/union/class or an enumeration that is defined in one module can be referenced or used in other modules. Visibility attributes affect only types with external linkage. By using this feature, you can make a shared library smaller and decrease the possibility of symbol collision. For details, see Using visibility attributes in the XL C/C++ Optimization and Programming Guide.

visibility type attribute syntax

```
►►──__attribute__──((──┬──visibility──────┬──(──┬──"default"────┬──)──))──────────►◄
                       └──__visibility__──┘      ├──"protected"──┤
                                                 ├──"hidden"─────┤
                                                 └──"internal"───┘
```

Example

In the following example, the visibility attribute of class A is protected, and that of enumeration E is hidden:

```
class __attribute__((visibility("protected"))) A {};
enum __attribute__((visibility("hidden"))) E {e1,e2} e;
```

Related reference:

"The visibility variable attribute" on page 37

visibility

"The visibility namespace attribute" on page 52

See Using visibility attributes in the XL C/C++ Optimization and Programming Guide

See -fvisibility (-qvisibility) in the XL C/C++ Compiler Reference

See -shared (-qmkshrobj) in the XL C/C++ Compiler Reference

See #pragma GCC visibility push (visibility), #pragma GCC visibility pop in the XL C/C++ Compiler Reference

Chapter 3. Standard features

The compiler fully supports the following language standards:

- C++11
- C++03
- C++98
- C11
- C99
- C89

Note: The compiler fully supports `_Thread_local` and partially supports atomics, which are optional C11 features. For atomics, the compiler does not support the following data types:

- Atomic floating point types
- Atomic complex and vector types
- Atomic structure types that are packed through pragmas, attributes, or options

Besides these standards, the compiler also supports the following C++14 features:

C++14 features

Note: IBM supports selected features of C++14 standard. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++14 features is complete, including the support of a new C++14 standard library, the implementation might change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++14 features.

- Polymorphic lambda expressions
- Variable templates

Note: Compiler support for language features that have runtime library requirements is dependent on the GCC runtime library on the Linux distribution.

Notices

Programming interfaces: Intended programming interfaces allow the customer to write programs to obtain the services of IBM XL C/C++ for Linux.

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, MD-NC119
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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, 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 websites are not part of the materials for this IBM product and use of those websites 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 want 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:

Intellectual Property Dept. for Rational Software
IBM Corporation
5 Technology Park Drive
Westford, MA   01886
U.S.A.

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.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating

platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2015.

PRIVACY POLICY CONSIDERATIONS:

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, or to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

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.

Adobe and the Adobe logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Index

Special characters

__VA_ARGS__ 38
_thread storage class specifier 28
$ 8
[ ] (vector subscript operator) 45

A

alias function attribute 10
alignment 33, 35
   structures 33
allocation
   uninitialized global variables 34
always_inline function attribute 11
arguments
   macro 38
   trailing 38
arrays
   flexible array member 8
asm
   statements 19
assembly
   statements 19

B

basic example, described viii
bool 46

C

cast expressions 42
   union type 48
   vector literal 42
compatibility
   XL C and GCC 6, 48
   XL C/C++ and GCC 1
   XL C++ and C11 51
   XL C++ and C99 1, 50
   XL C++ and GCC 51
compound
   literal 8
const 46
const function attribute 11

D

data types
   vector 46
declarations
   vector types 46
dollar sign 8

E

ellipsis
   in macro argument list 38

examples

examples
   inline assembly statements 23

F

file inclusion 27
flexible array member 8
format function attribute 12
function attribute
   alias 10
   always_inline 11
   const 11
   constructor 12
   destructor 12
   format 12
   format_arg 13
   noinline 16
   noreturn 16
   pure 16
   section 16
   weak 17
function attributes 9
function-like macro 38
functions
   specifiable attributes 9

G

global variable
   uninitialized 34

I

identifiers 8
include_next preprocessor directive 27
incomplete type
   as structure member 8
initialization
   order of 52
   static object 8
   vector types 39
initializer lists 8, 39
initializers
   vector types 39
inline
   assembly statements 19
integer
   literals 4

L

language extensions 1
linkage
   weak symbols 37
literals
   compound 8
   integer 4
   vector 42

long long

long long
   types of integer literals in C99 and
      C++11 4
   types of integer literals outside of C99
      and C++11 4
long long type specifier 46

M

macro
   function-like 38
   variable argument 38

O

operators
   [] (vector subscripting) 45

P

packed
   variable attribute 35
pixel 46
pointers
   vector types 40

S

statements
   inline assembly
      restrictions 23
static 46
storage class specifiers
   _thread 28
   tls_model attribute 28
structures
   flexible array member 8
   members
      incomplete types 8
subscripting operator 45
suffix
   integer literal constants 4

T

tls_model attribute 37
type attributes 29
   aligned 30
   may_alias 31
   packed 32
   transparent_union 49
type specifiers
   overriding 35
   vector data types 46

U

unions
    cast to union type   48

V

variable attributes   32
    section   36
vector
    binary operators   41
    literals   42
    subscripting operator   45
    unary operators   41
vector data types   46
vector literal
    cast expressions   42
vector processing support   1, 39, 50
vector types
    cast   46
    literals   42
    pointer arithmetic   40
visibility attributes
    class   53
    enumeration   53
    function   18
    namespace   52
    structure   53
    union   53
    variable   37

W

weak symbol   37

IBM®

Product Number: 5765-J08; 5725-C73

Printed in USA