IBM i
7.4

Programming
IBM Rational Development Studio for i
ILE C/C++ Compiler Reference

SC09-4816-08

Note

Before using this information and the product it supports, read the information in “Notices” on page 121.

This edition applies to IBM® Rational® Development Studio for i (product number 5770-WDS) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

This document may contain references to Licensed Internal Code. Licensed Internal Code is Machine Code and is licensed to you under the terms of the IBM License Agreement for Machine Code.

Contents

ILE C/C++ Compiler Reference

This information is for programmers who are familiar with the C and C++ programming languages and who plan to use the ILE C/C++ compiler to build new or maintain existing ILE C/C++ applications.

What is new for IBM i 7.3

Read about new or significantly changed information.

- New ILE C/C++ predefined macros. See “ILE C/C++ Predefined Macros” on page 12.
- New LANGLVL(*EXTENDED0X) Control Language Command option value. See “LANGLVL” on page 80.

PDF file for ILE C/C++ Compiler Reference

You can view and print a PDF file of this information.

To view or download the PDF version of this document, select ILE C/C++ Compiler Reference.

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html)  .

About ILE C/C++ Compiler Reference

Read this section for an overview of the compiler reference information.

You need experience in using applicable IBM i menus and displays or Control Language (CL) commands. You also need knowledge of ILE as explained in *ILE Concepts*.

Prerequisite and Related Information

Use the IBM i Information Center as your starting point for looking up IBM i technical information. You can access the Information Center from the following web site:

```
http://www.ibm.com/systems/i/infocenter
```

The IBM i Information Center contains new and updated system information, such as software installation, Linux®, WebSphere®, Java™, high availability, database, logical partitions, CL commands, and system application programming interfaces (APIs). In addition, it provides advisors and finders to assist in planning, troubleshooting, and configuring your system hardware and software.

For other related information, see the "Related information" on page 119.

Install Licensed Program Information

On systems that will be making use of the ILE C/C++ compiler, the QSYSINC library must be installed.

A Note About Examples

Examples illustrating the use of the ILE C/C++ compiler are written in a simple style. The examples do not demonstrate all of the possible uses of C or C++ language constructs. Some examples are only code fragments and do not compile without additional code.

Control Language Commands

If you need prompting, type the CL command and press F4 (Prompt). If you need online help information, press F1 (Help) on the CL command prompt display. CL commands can be used in either batch or interactive mode, or from a CL program.

For more information about CL commands, see the *CL and APIs* section in the *Programming* category at the IBM i Information Center web site:

```
http://www.ibm.com/systems/i/infocenter
```

You need object authority to use CL commands. For more information about object authority, see the *Planning and setting up system security* section in the *Security* category at the Information Center Web site.

How to Read the Syntax 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.

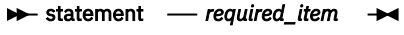
  The ►—— symbol indicates that a command, directive, or statement is continued from the previous line.

  The ——►◄ symbol indicates the end of a command, directive, or statement.

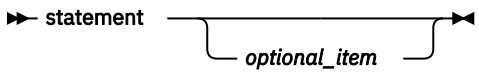
Diagrams of syntactical units other than complete commands, directives, or statements start with the ►—— symbol and end with the ——► symbol.

Note : In the following diagrams, `statement` represents a C or C++ command, directive, or statement.

- Required items appear on the horizontal line (the main path).

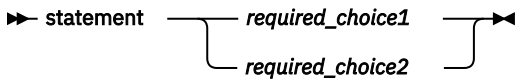


- Optional items appear below the main path.

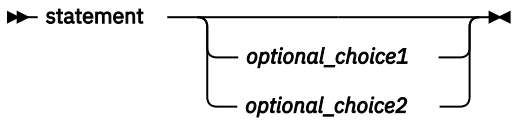


- If you can choose from two or more items, they appear vertically, in a stack.

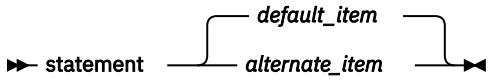
  If you *must* choose one of the items, one item of the stack appears on the main path.



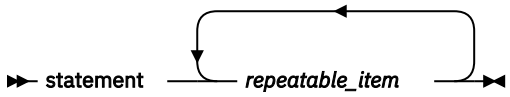
  If choosing one of the items is optional, the entire stack appears below the main path.



  The item that is the default appears above the main path.



- An arrow returning to the left above the main line indicates an item that can be repeated.



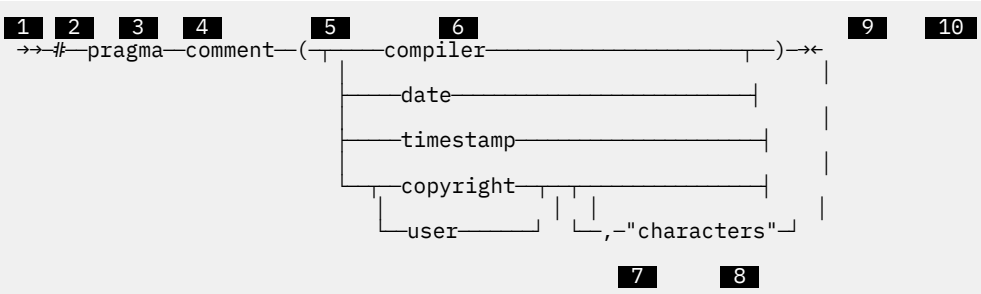
  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, `extern`).

  Variables appear in italicized lowercase letters (for example, *identifier*). 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.

The following syntax diagram example shows the syntax for the `#pragma comment` directive. See "ILE C/C++ Pragmas" on page 19 for information about the `#pragma` directive.



1 This is the start of the syntax diagram.

2 The symbol # must appear first.
3 The keyword `pragma` must appear following the # symbol.
4 The keyword `comment` must appear following the keyword `pragma`.
5 An opening parenthesis must be present.
6 The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`.
7 A comma must appear between the comment type `copyright` or `user`, and an optional character string.
8 A character string must follow the comma. The character string must be enclosed in double quotation marks.
9 A closing parenthesis is required.
10 This is the end of the syntax diagram.

The following examples of the `#pragma comment` directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

Industry Standards

The Integrated Language Environment® C/C++ compiler and runtime library are designed according to the following standards:

- 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 14882:1998, also known as C++98
- Information Technology - Programming languages - C++, ISO/IEC 14882:2003(E), also known as Standard C++
- Information Technology - Programming languages - Extension for the programming language C to support decimal floating-point arithmetic, ISO/IEC WDTR 24732. This draft technical report has been submitted to the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1176.pdf.

ILE C supports a subset of C99 features.

ILE C++ supports a subset of C++0x features.

Note : C++0x is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++0x standard is complete, including the support of a new C++ standard library, the implementation may 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 features of the C++0x standard and therefore they should not be relied on as a stable programming interface.

C++0x has been ratified and published as ISO/IEC 14882:2011. All references to C++0x in this document are equivalent to the ISO/IEC 14882:2011 standard. Corresponding information, including programming interfaces, will be updated in a future release.

Predefined Macros

Several predefined macros are recognized. Read this section for details on predefined macros.

The ILE C/C++ compiler recognizes the following predefined macros.

ANSI/ISO Standard Predefined Macros

The ILE C/C++ compiler recognizes the following macros defined by the ANSI/ISO Standard. Unless otherwise specified, macros when defined have a value of 1.

__DATE__
A character string literal containing the date when the source file was compiled. The date is in the form:

```
"Mmm dd yyyy"
```

where:

- Mmm represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).
- dd represents the day. If the day is less than 10, the first d is a blank character.
- yyyy represents the year.

__FILE__
Defined as a character string literal containing the name of the source file.

__LINE__
Defined to be an integer representing the current source line number.

__STDC__
C Defined if the C compiler conforms to the ANSI standard. This macro is defined if the language level is set to LANGLVL(*ANSI).

__STDC_VERSION__
C Defined to be an integer constant of type long int. This macro is defined only if __STDC__ is also defined and has the value 199409L. This macro is not defined for C++.

__TIME__
Defined as a character string literal containing the time when the source file was compiled. The time is in the form:

```
"hh:mm:ss"
```

where:

- hh represents the hour.
- mm represents the minutes.
- ss represents the seconds.

__cplusplus
C++ Defined when compiling a C++ program, indicating that the compiler is a C++ compiler. This macro has no trailing underscores. This macro is not defined for C.

Note :

1. Predefined macro names cannot be the subject of a #define or #undef preprocessor directive.

2. The predefined ANSI/ISO Standard macro names consist of two underscore (__) characters immediately preceding the name, the name in uppercase letters, and two underscore characters immediately following the name.
3. The value of __LINE__ changes during compilation as the compiler processes subsequent lines of your source program.
4. The value of __FILE__ and __TIME__ changes as the compiler processes any #include files that are part of your source program.
5. C You can also change __LINE__ and __FILE__ using the #line preprocessor directive.

Examples

The following printf() statements display the values of the predefined macros __LINE__, __FILE__, __TIME__, and __DATE__ and print a message indicating if the program conforms to ANSI standards based on __STDC__:

```
#include <stdio.h>
#ifdef __STDC__
#   define CONFORM    "conforms"
#else
#   define CONFORM    "does not conform"
#endif
int main(void)
{
  printf("Line %d of file %s has been executed\n", __LINE__, __FILE__);
  printf("This file was compiled at %s on %s\n", __TIME__, __DATE__);
  printf("This program %s to ANSI standards\n", CONFORM);
}
```

Related Information

See the *ILE C/C++ Language Reference* for additional information on predefined macros.

ILE C/C++ Predefined Macros

The ILE C/C++ compiler provides the predefined macros described in this section. These macros are defined when their corresponding pragmas are invoked in program source, or when their corresponding compiler options are specified. Unless otherwise specified, macros when defined have a value of 1.

__ANSI__
Defined when the LANGLVL(*ANSI) compiler option is in effect. When this macro is defined, the compiler allows only language constructs that conform to the ANSI/ISO C and C++ standards.

__ASYNC_SIG__
C Defined when the SYSIFCOPT(*ASYNCSIGNAL) compiler option is in effect.

C++ Defined when TERASPACE(*YES *TSIFC) STGMDL(*TERASPACE) DTAMDL(*LLP64) RTBND(*LLP64) compiler options are in effect.

__BASE_FILE__
Indicates the fully qualified name of the primary source file.

__BOOL__
C++ Indicates that the bool keyword is accepted.

_CHAR_SIGNED __CHAR_SIGNED__
Defined when the #pragma chars(signed) directive is in effect, or when the DFTCHAR compiler option is set to *SIGNED. If this macro is defined, the default character type is signed.

_CHAR_UNSIGNED __CHAR_UNSIGNED__
Defined when the #pragma chars(unsigned) directive is in effect, or when the DFTCHAR compiler option is set to *UNSIGNED. If this macro is defined, indicates default character type is unsigned.

__cplusplus98__interface__
C++ Defined when the LANGLVL(*ANSI) compiler option is specified.

__C99_BOOL

 Indicates support for the _Bool data type. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_CPLUSCMT

 Indicates support for C++ style comments. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_COMPOUND_LITERAL

Indicates support for compound literals.

 Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

 Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__C99_DESIGNATED_INITIALIZER

 Indicates support for designated initialization. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_DUP_TYPE_QUALIFIER

 Indicates support for duplicated type qualifiers. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_EMPTY_MACRO_ARGUMENTS

 Indicates support for empty macro arguments. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_FLEXIBLE_ARRAY_MEMBER

 Indicates support for flexible array members. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99__FUNC__

Indicates support for the __func__ predefined identifier.

 Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

 Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__C99_HEX_FLOAT_CONST

Indicates support for hexadecimal floating constants.

 Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

 Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__C99_INLINE

 Indicates support for the inline function specifier. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_LLONG

 Indicates support for C99-style long long data types and literals. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_MACRO_WITH_VA_ARGS

Indicates support for function-like macros with variable arguments.

 Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

 Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__C99_MAX_LINE_NUMBER
Indicates that the maximum line number is 2147483647.

C Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

C++ Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__C99_MIXED_DECL_AND_CODE
C Indicates support for mixed declaration and code. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_MIXED_STRING_CONCAT
C++ Indicates support for concatenation of wide string and non-wide string literals. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__C99_NON_CONST_AGGR_INITIALIZER
C Indicates support for non-constant aggregate initializers. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_NON_LVALUE_ARRAY_SUB
C Indicates support for non-lvalue subscripts for arrays. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_PRAGMA_OPERATOR
Indicates support for the _Pragma operator.

C Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

C++ Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__C99_RESTRICT
C++ Indicates support for the C99 restrict qualifier. Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__C99_STATIC_ARRAY_SIZE
C Indicates support for the static keyword in array parameters to functions. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_VAR_LEN_ARRAY
C Indicates support for variable length arrays. Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

__C99_VARIABLE_LENGTH_ARRAY
C++ Indicates support for variable length arrays. Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__DIGRAPHS__
Indicates support for digraphs.

__EXTENDED__
C Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

C++ Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

When this macro is defined, the compiler allows language extensions provided by the ILE C/C++ compiler implementation.

__FUNCTION__
Indicates the name of the function currently being compiled. For C++ programs, expands to the actual function prototype.

__HHW_AS400__
Indicates that the host hardware is an IBM i processor.

__HOS_OS400__

C++ Indicates that the host operating system is IBM i.

__IBMC__

C Indicates the version of the C compiler. It returns an integer of the form VRM where:

V represents the version

R represents the release

M represents the modification level

For example, using the IBM i 7.3 compiler with the TGTRLS(*CURRENT) compiler option, __IBMC__ returns the integer value 730.

__IBMCPP__

C++ Indicates the version of the AIX® XL C++ compiler upon which the ILE C++ compiler is based. It returns an integer representing the compiler version. For example, using the IBM i 7.3 compiler with the TGTRLS(*CURRENT) compiler option, __IBMCPP__ returns the integer value 1310. 1310 means the ILE C++ compiler is based on the XL C++ V13.1 compiler.

__IBM__ALIGN

C++ Indicates support for the __align specifier.

__IBM_ATTRIBUTES

C++ Indicates support for type, variable, and function attributes. Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBM_COMPUTED_GOTO

C++ Indicates support for computed goto statements. Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBM_DFP__

Indicates support for decimal floating-point types.

C Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

C++ Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBM_EXTENSION_KEYWORD

C++ Indicates support for the __extension__ keyword. Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBM_INCLUDE_NEXT

Indicates support for the `#include_next` preprocessing directive.

__IBM_LABEL_VALUE

C++ Indicates support for labels as values. Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBM_LOCAL_LABEL

C++ Indicates support for local labels. Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBM_MACRO_WITH_VA_ARGS

C++ Indicates support for variadic macro extensions. Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBM_TYPEOF__

Indicates support for the __typeof__ or typeof keyword. This macro is always defined for C.

For C++, it is defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_AUTO_TYPEDEDUCTION

C++ Indicates support for the auto type deduction feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_CONSTEXPR

C++11 Indicates support for the generalized constant expressions feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_C99_PREPROCESSOR

C++ Indicates support for the C99 preprocessor features adopted in the C++0x standard. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_DECLTYPE

C++ Indicates support for the decltype feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_DEFAULTED_AND_DELETED_FUNCTIONS

C++11 Indicates support for the defaulted and deleted functions feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_DELEGATING_CTORS

C++ Indicates support for the delegating constructors feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_EXPLICIT_CONVERSION_OPERATORS

C++11 Indicates support for the explicit conversion operators feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_EXTENDED_FRIEND

C++ Indicates support for the extended friend declarations feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_EXTERN_TEMPLATE

C++ Indicates support for the explicit instantiation declarations feature. Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_INLINE_NAMESPACE

C++ Indicates support for the inline namespace definitions feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_NULLPTR

C++11 Indicates support for the nullptr feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_REFERENCE_COLLAPSING

C++11 Indicates support for the reference collapsing feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_RIGHT_ANGLE_BRACKET

C++ Indicates support for the right angle bracket feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_RVALUE_REFERENCES

C++11 Indicates support for the rvalue references feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_SCOPED_ENUM

C++11 Indicates support for the scoped enumeration feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_STATIC_ASSERT

C++ Indicates support for the static assertions feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IBMCPP_VARIADIC_TEMPLATES

Indicates support for the variadic templates feature. Defined when the LANGLVL(*EXTENDED0X) compiler option is in effect.

__IFS_IO__

Defined when the SYSIFCOPT(*IFSIO) or SYSIFCOPT(*IFS64IO) compiler option is specified.

__IFS64_IO__

Defined when the SYSIFCOPT(*IFS64IO) compiler option is specified. When this macro is defined, _LARGE_FILES and _LARGE_FILE_API are also defined in the relevant IBM-supplied header files.

__ILEC400__

Indicates that the ILE C compiler is being used.

__ILEC400_TGTVRM__

Same as the __OS400_TGTVRM__ macro.

_LARGE_FILES

Defined when the SYSIFCOPT(*IFS64IO) compiler option is in effect and system header file types.h is included.

_LARGE_FILE_API

Defined when the SYSIFCOPT(*IFS64IO) compiler option is in effect and system header file types.h is included.

__LLP64_IFC__

Defined when the DTAMDL(*LLP64) compiler option is in effect.

__LLP64_RTBND__

Defined when the RTBND(*LLP64) compiler option is in effect.

__LONGDOUBLE64

Indicates that the size of a long double type is 64 bits. This macro is always defined.

_LONG_LONG

Indicates support for IBM long long data types.

Defined when the LANGLVL(*EXTENDED) compiler option is in effect.

Defined when the LANGLVL(*EXTENDED) or LANGLVL(*EXTENDED0X) compiler option is in effect.

__NO_RTTI__

Defined when the OPTION(*NORTTI) compiler option is in effect.

__OPTIMIZE__

Indicates the level of optimization in effect. The macro is undefined for OPTIMIZE(10). For other OPTIMIZE settings, the macro is defined as follows:

2 for OPTIMIZE(20)

3 for OPTIMIZE(30)

4 for OPTIMIZE(40)

__OS400__

This macro is always defined when the compiler is used with the IBM i operating system.

__OS400_TGTVRM__

Defines an integer value that maps to the version/release/modification level of the operating system that the generated object is intended to run on. For example, if the target release is set using the TGTRLS(V7R2M0) compiler option, this macro returns the integer value 720.

__POSIX_LOCALE__

Defined when the LOCALETYPE(*LOCALE) or LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) compiler option is specified.

__RTTI_DYNAMIC_CAST__
C++ Defined when the OPTION(*RTTIALL) or OPTION(*RTTICAST) compiler option is specified.

__RTTI_TYPE_INFO__
C++ Defined when the OPTION(*RTTIALL) or OPTION(*RTTITYPE) compiler option is specified.

__SIZE_TYPE__
C Indicates the underlying type of size_t on the current platform. For IBM i, it is unsigned int.

__SRCSTMF__
C Defined when the SRCSTMF compiler option specifies the location of the source file being compiled.

__TERASPACE__
Defined when the TERASPACE(*YES *TSIFC) compiler option is specified.

__THW_AS400__
Indicates that the target hardware is an IBM i processor.

__TIMESTAMP__
A character string literal containing the date and time when the source file was last changed.

The date and time are in the form:

```
"Day Mmm dd hh:mm:ss yyyy"
```

where:

`Day` represents the day of the week (`Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`, or `Sun`).

`Mmm` represents the month in an abbreviated form (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`).

`dd` represents the day. If the day is less than 10, the first d is a blank character.

`hh` represents the hour.

`mm` represents the minutes.

`ss` represents the seconds.

`yyyy` represents the year.

Note : Other compilers might not support this macro. If the macro is supported on other compilers, the date and time values might be different from the values shown here.

__TOS_OS400__
Indicates that the target operating system is IBM i.

__UCS2__
Defined when the LOCALETYPE(*LOCALEUCS2) compiler option is specified.

__UTF32__
Defined when the LOCALETYPE(*LOCALEUTF) compiler option is specified.

__wchar_t
Indicates that the typedef wchar_t has been defined.

C This macro is defined in the standard header file stddef.h.

C++ This macro is defined by the C++ compiler.

ILE C/C++ Pragmas

Read this section for an overview of pragma directives.

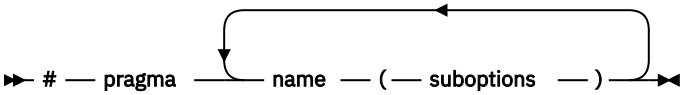
Pragma directive syntax

There are two forms of pragma directives supported in the ILE C/C++ compilers:

`#pragma name`

This form uses the following syntax:

#pragma name syntax

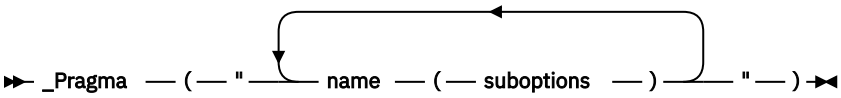


The *name* is the pragma directive name, and the *suboptions* are any required or optional suboptions that can be specified for the pragma, where applicable.

`_Pragma ("name")`

This form uses the following syntax:

_Pragma("name") syntax



For example, the statement:

```
_Pragma ("pack(1)")
```

is equivalent to:

```
#pragma pack(1)
```

For all forms of pragma statements, you can specify more than one *name* and *suboptions* in a single `#pragma` statement.

The *name* on a pragma is subject to macro substitutions, unless otherwise stated. The compiler ignores unrecognized pragmas, issuing an informational message (C++) or warning message (C) indicating the unrecognized pragma.

If you have any pragmas that are not common to both C and C++ in code that is compiled by both compilers, you should add conditional compilation directives around the pragmas. (These directives are not strictly necessary since unrecognized pragmas are ignored.) For example, `#pragma info` is only recognized by the C++ compiler, so you might decide to add conditional compilation directives around the pragma.

```
#ifdef __cplusplus
#pragma info(none)
#endif
```

Scope of Pragma directives

Many pragma directives can be specified at any point within the source code in a compilation unit; others must be specified before any other directives or source code statements. The individual descriptions for each pragma describe any constraints on the placement of the pragma.

In general, if you specify a pragma directive before any code in your source program, it applies to the entire compilation unit, including any header files that are included. For a directive that can appear anywhere in your source code, it applies from the point at which it is specified, until the end of the compilation unit.

You can further restrict the scope of the application of a pragma by using complementary pairs of pragma directives around a selected section of code. For example, using `#pragma datamodel` directives as follows requests that only the selected parts of your source code use a particular data model setting:

```
/* Data model may be P128 or LLP64 */
#pragma datamodel(P128)
/* Data model P128 is now in effect */
#pragma datamodel(pop)
/* Prior data model is now in effect */
```

Many pragma directives provide "pop“ or "reset" suboptions that allow you to enable and disable pragma settings in a stack-based fashion; examples of these suboptions are provided in the relevant pragma descriptions.

Summary of Pragma Directives

The ILE C/C++ compiler recognizes the following pragmas:

Table 1. Pragmas Recognized by the ILE C/C++ Compiler

Pragma Name	Valid with C	Valid with C++
“argopt” on page 21	Yes	Yes
“argument” on page 23	Yes	No
“cancel_handler” on page 24	Yes	Yes
“chars” on page 25	Yes	Yes
“checkout” on page 25	Yes	No
“comment” on page 26	Yes	Yes
“convert” on page 27	Yes	No
“datamodel” on page 27	Yes	Yes
“define” on page 28	No	Yes
“descriptor” on page 29	Yes	Yes
“disable_handler” on page 30	Yes	Yes
“disjoint” on page 30	No	Yes
“do_not_instantiate” on page 31	No	Yes
“enum” on page 32	Yes	Yes
“exception_handler” on page 36	Yes	Yes
“hashome” on page 39	No	Yes
“implementation” on page 39	No	Yes
“info” on page 39	No	Yes
“inline” on page 41	Yes	No
“ishome” on page 41	No	Yes

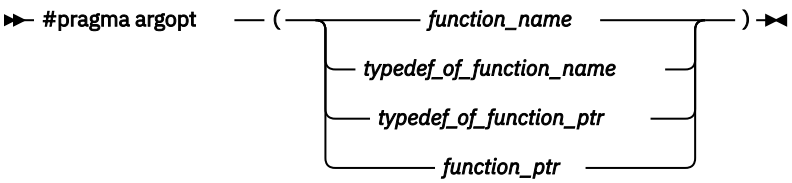
Table 1. Pragmas Recognized by the ILE C/C++ Compiler (continued)		
Pragma Name	**Valid with C**	**Valid with C++**
“isolated_call” on page 42	No	Yes
“linkage” on page 42	Yes	No
“map” on page 44	Yes	Yes
“mapinc” on page 45	Yes	No
“margins” on page 47	Yes	No
“namemangling” on page 47	No	Yes
“namemanglingrule” on page 48	No	Yes
“noargv0” on page 49	Yes	No
“noinline (function)” on page 50	Yes	No
“nomargins” on page 50	Yes	No
“nosequence” on page 50	Yes	No
“nosigtrunc” on page 50	Yes	No
“pack” on page 51	Yes	Yes
“page” on page 56	Yes	No
“pagesize” on page 56	Yes	No
“pointer” on page 56	Yes	Yes
“priority” on page 57	No	Yes
“sequence” on page 58	Yes	No
“strings” on page 59	Yes	Yes
“weak” on page 59	No	Yes

Individual Pragma Descriptions

argopt

C C++

argopt syntax



Description

Argument Optimization (argopt) is a pragma which might improve runtime performance. Applied to a bound procedure, optimizations can be achieved by:

- Passing space pointer parameters into general-purpose registers (GPRs).
- Storing a space pointer returned from a function into a GPR.

Parameters

function_name
Specifies the name of the function for which optimized procedure parameter passing is to be specified. The function can be either a static function, an externally-defined function, or a function defined in the current compilation unit that is called from outside the current compilation unit.

typedef_of_function_name
Specifies the name of the typedef of the function for which optimized procedure parameter passing is to be specified.

typedef_of_function_ptr
Specifies the name of the typedef of the function pointer for which optimized procedure parameter passing is to be specified.

function_ptr
Specifies the name of the function pointer for which optimized procedure parameter passing is to be specified.

Notes on Usage

Specifying #pragma argopt directive does not guarantee that your program will be optimized. Participation in argopt is dependent on the translator.

Do not specify #pragma argopt together with #pragma descriptor for the same declaration. The compiler supports using only one or the other of these pragmas at a time.

A function must be declared (prototyped), or defined before it can be named in a #pragma argopt directive.

Void pointers will not be optimized since they are not space pointers.

Use of #pragma argopt is not supported in struct declarations.

The #pragma argopt cannot be specified for functions which have OS-linkage or built-in linkage (for functions which have a #pragma linkage (function_name, OS) directive or #pragma linkage(function_name, builtin) directive associated with them, and vice versa).

The #pragma argopt will be ignored for functions which are named as handler functions in #pragma exception_handler or #pragma cancel_handler directives, and error handling functions such as `signal()` and `atexit()`. The #pragma argopt directive cannot be applied to functions with a variable argument list.

#pragma argopt scoping

The #pragma argopt must be placed in the same scope as the function, the function pointer, typedef of a function pointer, or typedef of a function that it operates on. If the #pragma argopt is not in the same scope, an error is issued.

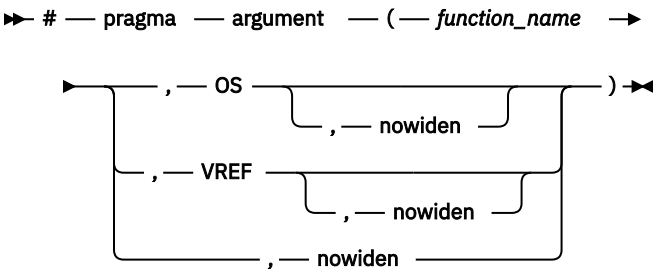
```
#include <stdio.h>

long func3(long y)
{
printf("In func3()\n");
printf("hex=%x,integer=%d\n",y, y);
}
#pragma argopt (func3)           /* file scope of function  */
int main(void)
 {
 int i, a=0;
 typedef long (*func_ptr) (long);
 #pragma argopt (func_ptr)      /* block scope of typedef   */
                                /* of function pointer      */
 struct funcstr
    {
     long (*func_ptr2) (long);
     #pragma argopt (func_ptr2) /* struct scope of function */
                                /* pointer                  */
     };
struct funcstr func_ptr3;
for (i=0; i<99; i++)
 {
```

```
  a = i*i;
  if (i == 7)
    {
     func_ptr3.func_ptr2( i );
    }
 }
 return i;
}
```

argument

C

argument syntax



Description

Specifies the argument passing and receiving mechanism to be used for the procedure or typedef named by *function_name*.

This pragma identifies procedures as externally bound-procedures only. The procedure may be defined in and called from the same source as the pragma argument directive. If the pragma argument directive is specified in the same compilation unit as the definition of the procedure named in that directive, the arguments to that procedure will be received using the method specified in that pragma directive.

For information about making calls to external programs, see pragma "linkage" on page 42 .

Parameters

function_name
Specifies the name of the externally-bound procedure.

OS
OS indicates that arguments are passed, or received (if the pragma directive is in the same compilation unit as the procedure definition), using the OS-Linkage argument method. Non-address arguments are copied to temporary locations and widened (unless nowiden has been specified), and the address of the copy is passed to the called procedure. Arguments that are addresses or pointers are passed directly to the called procedure.

VREF
VREF is similar to OS-linkage with the exception that address arguments are also passed and received using the OS-Linkage method.

nowiden
Specifies that the arguments are not widened before they are passed or received. This parameter can be used by itself without specifying an argument type. For example, `#pragma argument (myfunc, nowiden)`, indicates that procedure myfunc will pass and receive its arguments with the typical *by-value* method, but unwidened.

Notes on Usage

This pragma controls how parameters are passed to bound-procedures and how they are received. The function name specified in the #pragma argument directive can be defined in the current compilation unit. The #pragma argument directive must precede the function it names.

Specifying a #pragma argument directive in the same compilation unit as the affected procedure tells the compiler that the procedure is to receive (as well as to send) its arguments as specified in the pragma

argument directive. This is useful for ILE C written bound-procedures specified in a pragma argument. The user must ensure that if the call to the procedure and the definition are in separate compilation units, the pragma argument directives must match in regards to their passing method (OS, VREF, and nowiden).

For example, in the two source files below, the address of a temporary copy of the argument will be passed to `foo` in Program 1. Program 2, `foo` will receive the address of the temporary copy, dereference it, and assign that value to the parameter a. If the two pragma directives differ, behavior is undefined.

Program 1	Program 2

`#pragma argument(foo, OS, nowiden)`
`void foo(char);`
`void main() {`
`  foo(10);`
`}`
	`#pragma argument(foo, OS, nowiden)`
	`void foo(char a) { a++; }`

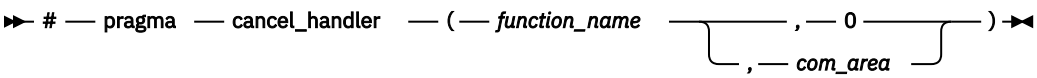
Warnings are issued, and the #pragma argument directive is ignored if any of the following occurs:

- The #pragma argument directive does not precede the declaration or definition of the named function in the compilation unit.
- The *function_name* in the directive is not the name of a procedure or a typedef of a procedure.
- A typedef named in the directive has been used in the declaration or definition of a procedure before being used in the directive.
- A #pragma argument directive has already been specified for this function.
- A #pragma linkage directive has already been specified for this function.
- The function has already been called before the #pragma argument directive.

cancel_handler

C C++

cancel_handler syntax



Description

Specifies that the function named is to be enabled as a user-defined ILE cancel handler at the point in the code where the #pragma cancel_handler directive is located.

Any cancel handler that is enabled by a #pragma cancel_handler directive is implicitly disabled when the call to the function containing the directive is finished. The call is removed from the call stack, if the handler has not been explicitly disabled by the #pragma disable_handler directive.

Parameters

function_name
Specifies the name of the function to be used as a user-defined ILE cancel handler.

com_area
Used to pass information to the exception handler. If no *com_area* is required, specify zero as the second parameter of the directive. If a *com_area* is specified on the directive, it must be a variable of one of the following data types: integral, float, double, struct, union, array, enum, pointer, or packed decimal. The com_area should be declared with the volatile qualifier. It cannot be a member of a structure or a union.

See the *C/C++ Runtime Library Functions* for information about <except.h> and the typedef _CNL_Hndlr_Parms_T, a pointer which is passed to the cancel handler.

Notes on Usage

The handler function can take only 16-byte pointers as parameters.

This #pragma directive can only occur at a C language statement boundary and inside a function definition.

The compiler issues an error message if any of the following occurs:

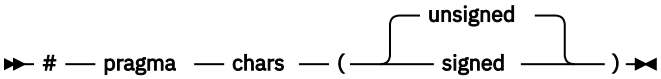
- The directive occurs outside a C function body or inside a C statement.
- The handler function is not declared or defined.
- The identifier that is named as the handler function is not a function.
- The *com_area* variable is not declared.
- The *com_area* variable does not have a valid object type.
- The handler function specified is defined with argument optimization (#pragma argopt).

See the *ILE C/C++ Programmer's Guide* for examples and more information about using the #pragma cancel_handler directive.

chars

C C++

chars syntax



Description

Specifies that the compiler is to treat all `char` objects as `signed` or `unsigned`. This pragma must appear before any C code or directive (except for the #line directive) in a source file.

Parameters

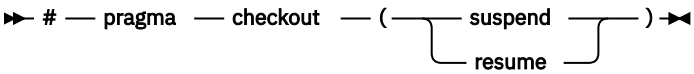
unsigned
All `char` objects are treated as `unsigned` integers.

signed
All `char` objects are treated as `signed` integers.

checkout

C

checkout syntax



Description

Specifies whether the compiler should give compiler information when a CHECKOUT compiler option value other than *NONE is specified.

Parameters

suspend
Specifies that the compiler suspend informational messages.

`resume`
Specifies that the compiler resume informational messages.

Notes on Usage

#pragma checkout directives can be nested. This means that a #pragma checkout (suspend) directive will have no effect if a previously specified #pragma checkout (suspend) directive is still in effect. This is also true for the #pragma checkout resume directive.

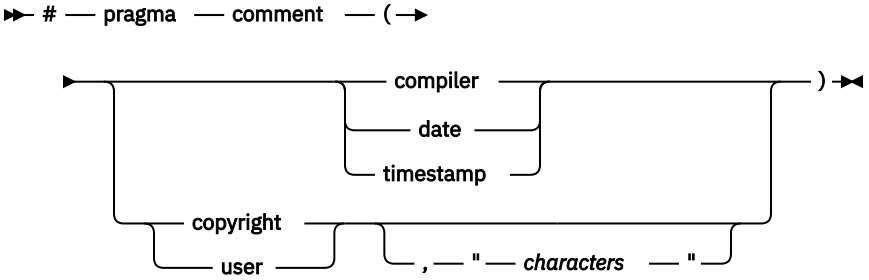
Example

```
/*  Assume CHECKOUT(*PPTRACE) had been specified                        */
#pragma checkout(suspend)  /* No CHECKOUT diagnostics are performed    */
   ...
#pragma checkout(suspend)  /* No effect                                */
   ...
#pragma checkout(resume)   /* No effect                                */
   ...
#pragma checkout(resume)   /* CHECKOUT(*PPTRACE) diagnostics continue  */
```

comment

C C++

comment syntax



Description

Emits a comment into the program or service program object. This can be shown by DSPPGM or DSPSRVPGM with DETAIL(*COPYRIGHT). This pragma must appear before any C code or directive (except for the #line directive) in a source file.

Parameters

Valid settings for the `comment` pragma can be:

compiler
The name and version of the compiler is emitted into the end of the generated program object.

date
The date and time of compilation is emitted into the end of the generated program object.

timestamp
The last modification date and time of the source is emitted into the end of the generated program object.

copyright
The text that is specified by *characters* is placed by the compiler into the generated program object and is loaded into memory when the program is run.

user
The text specified by *characters* is placed by the compiler into the generated object. However, it is not loaded into memory when the program is run.

Notes on Usage

The copyright and user comment types are virtually the same for the ILE C/C++ compiler. One has no advantage over the other.

The maximum number of characters in the text portion of a #pragma comment(copyright) or #pragma comment(user) directive is 256.

The maximum number of #pragma comment directives that can appear in a single compilation unit is 1024.

convert

C C++

convert syntax

Description

Specifies the Coded Character Set Identifier (CCSID) to use for converting the string literals from that point onward in a source file during compilation. The conversion continues until the end of the source file or until another #pragma convert directive is specified. Use #pragma convert (0) to disable the previous #pragma convert directive. The CCSID of the string literals before conversion is the same CCSID as the root source member. CCSIDs 905 and 1026 are not supported. The CCSID can be either EBCDIC or ASCII.

Parameters

ccsid
Specifies the coded character set identifier to use for converting the strings and literals in the source file. The value can be 0 - 65535. See *ILE C/C++ Runtime Library Functions* for more information about code pages.

Notes on Usage

By default, runtime library functions that parse format strings (such as `printf()` and `scanf()`) expect the format strings to be coded in CCSID 37. If the LOCALETYPE(*LOCALEUTF) compile option is specified, then the runtime library functions expect the format strings to be coded in the CCSID of the last locale set in the program (or UTF-8 if the program does not set the locale).

String and character constants that are specified in hex, for example (0xC1), are not converted.

Substitution characters are not used when converting to a target CCSID that does not contain the same symbol set as the source CCSID. The compilation fails.

If a CCSID with the value 65535 is specified for the C compiler, it behaves the same as if a value of 0 is specified. For the C++ compiler the CCSID of the root source member is assumed. If the source file CCSID value is 65535, the job CCSID is assumed for the source file. If the file CCSID is 65535 and the job CCSID is not 65535, the job CCSID is assumed for the file CCSID. If the file CCSID is 65535 and the job CCSIDis also 65535, but the system CCSID value is not 65535, the system CCSID value is assumed for the file CCSID. If the file, job and system CCSID values are 65535, CCSID 037 is assumed.

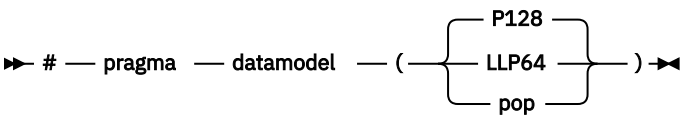
For include processing, the CCSID at the start of the header is the CCSID in effect at the point of the #include. At the end of the header file, the CCSID is changed back to the CCSID that was in effect at the point of the #include. Good programming practice dictates that within a header file, any convert(ccsid) pragmas should have a corresponding convert(0) pragma before the end of the header file.

If the LOCALETYPE(*LOCALE) compiler option is specified for the C compiler, wide-character literals are not converted. For the C++ compiler, wide-character literals are converted to the code page requested by the convert pragma. If the LOCALETYPE(*LOCALEUTF) or LOCALETYPE(*LOCALEUCS2) compiler option is specified, wide-character literals are not converted. See *Using Unicode Support for Wide-Character Literals* in the *ILE C/C++ Programmer's Guide* for more information.

datamodel

C C++

datamodel syntax



Description

Specifies a data model to apply to a section of code. The data model setting determines the interpretation of pointer types in absence of an explicit modifier.

This pragma overrides the data model specified by the DTAMDL compiler command line option.

Parameters

P128, p128
The size of pointers declared without the __ptr64 keyword will be 16 bytes.

LLP64, llp64
The size of pointers declared without the __ptr128 keyword will be 8 bytes.

pop
Restores the previous data model setting. If there is no previous data model setting, the setting specifed by the DTAMDL compiler command line option is used.

Note on Usage

C++ This pragma and its settings are case-sensitive when used in C++ programs.

Specifying #pragma datamodel(LLP64) or #pragma datamodel(llp64) has effect only when the TERASPACE(*YES) compiler option is also specified.

The data model specified by this pragma remains in effect until another data model is specified, or until #pragma datamodel(pop) is specified.

Example

This pragma is recommended for wrapping header files, without having to add pointer modifiers to pointer declarations. For example:

```
// header file blah.h
#pragma datamodel(P128)     // Pointers are now 16-byte
char* Blah(int, char *);
#pragma datamodel(pop)      // Restore previous setting of datamodel
```

You can also specify data models using the __ptr64 and __ptr128 pointer modifiers. These modifers override the DTAMDL compiler option, and the #pragma datamodel setting for a specific pointer declaration.

The __ptr64 modifier should only be used if the TERASPACE(*YES) compiler option is also specified. The __ptr128 modifier can be used at any time.

The following example shows the declarations of a process local pointer and a tagged space pointer:

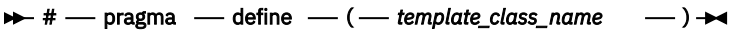
```
char * __ptr64   p;  // an 8-byte, process local pointer
char * __ptr128  t;  // a 16-byte, tagged space pointer
```

For more information, see *Using Teraspace* in the *ILE C/C++ Programmer's Guide*, and *Teraspace and single-level store* in the *ILE Concepts*.

define

C++

define syntax



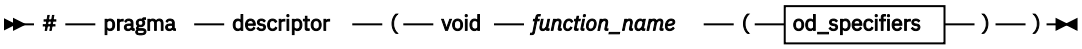
Description

The #pragma define directive forces the definition of a template class without actually defining an object of the class. The pragma can appear anywhere that a declaration is allowed. It is used when organizing your program for the efficient or automatic generation of template functions.

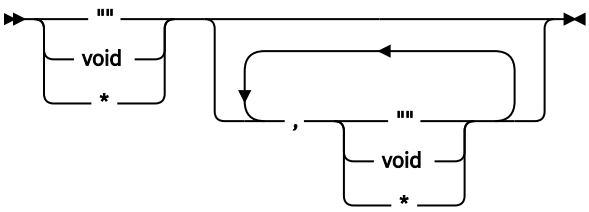
descriptor

C C++

descriptor syntax



od_specifiers



Description

An operational descriptor is an optional piece of information that is associated with a function argument. This information is used to describe an argument's attributes, for example, its data type and length. The #pragma descriptor directive is used to identify functions whose arguments have operational descriptors.

Operational descriptors are useful when passing arguments to functions that are written in other languages that may have a different definition of the data types of the arguments. For example, C defines a string as a contiguous sequence of characters ended by and including the first null character. In another language, a string may be defined as consisting of a length specifier and a character sequence. When passing a string from a C function to a function written in another language, an operational descriptor can be provided with the argument to allow the called function to determine the length and type of the string being passed.

The ILE C/C++ compiler generates operational descriptors for arguments that are passed to a function specified in a #pragma descriptor directive. The generated descriptor contains the descriptor type, data type, and length for each argument that is identified as requiring an operational descriptor. The information in an operational descriptor can be retrieved by the called function using the ILE APIs CEEGSI and CEEDOD. For more information about CL commands, see the *CL and APIs* section in the *Programming* category at the IBM i Information Center web site:

```
http://www.ibm.com/systems/i/infocenter
```

For the operational descriptor to determine the correct string length when passed through a function, the string has to be initialized.

The ILE C compiler supports operational descriptors for describing strings.

Note : A character string in ILE C/C++ is defined by using any one of the following ways:

- char string_name[n]
- char * string_name
- A string literal

Parameters

function_name
The name of the function whose arguments require operational descriptors.

od_specifiers
A list of symbols, that consists of "", void, or *, separated by commas, that specify which of a function's arguments are to have operational descriptors. An *od_specifier* list is similar to the argument list of a function except that an *od_specifier* list for a function can have fewer specifiers than its argument list.

- If a string operational descriptor is required for an argument, "" or * should be specified in the equivalent position for the *od_specifier* parameter.

- If an operational descriptor is not required for an argument then *void* is specified for that parameter in the equivalent position for the *od_specifier* list.

Notes on Usage

Do not specify #pragma descriptor together with #pragma argopt for the same declaration. The compiler supports using only one or the other of these pragmas at a time.

The compiler issues a warning and ignores the #pragma descriptor directive if any of the following conditions occur:

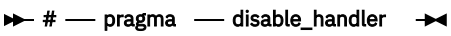
- The identifier specified in the pragma directive is not a function.
- The function is already specified in another pragma descriptor.
- The function is declared as static.
- The function has already been specified in a #pragma linkage directive.
- The function specified is a user entry procedure, for example, `main()`.
- The function is not prototyped before its #pragma descriptor directive.
- A call to the function occurs before its #pragma descriptor directive.

When using operational descriptors consider the following restrictions:

- Operational descriptors are only generated for functions that are called by their function name. Functions that are called by function pointer do not have operational descriptors generated.
- Operational descriptors are not allowed for C++ function declaration.
- If there are fewer *od_specifiers* than function arguments, the remaining *od_specifiers* default to void.
- If a function requires a variable number of arguments, the #pragma descriptor directive can specify that operational descriptors are to be generated for the required arguments but not for the variable arguments.
- It is not valid to do pointer arithmetic on a literal or array while it is also used as an argument that requires an operational descriptor, unless explicitly cast to char *. For example, if F is a function that takes as an argument a string, and F requires an operational descriptor for this argument, then the argument on the following call to F is not valid: F(a + 1) where "a" is defined as char a[10].

disable_handler

disable_handler syntax



Description

Disables the handler most recently enabled by either the exception_handler or cancel_handler pragma.

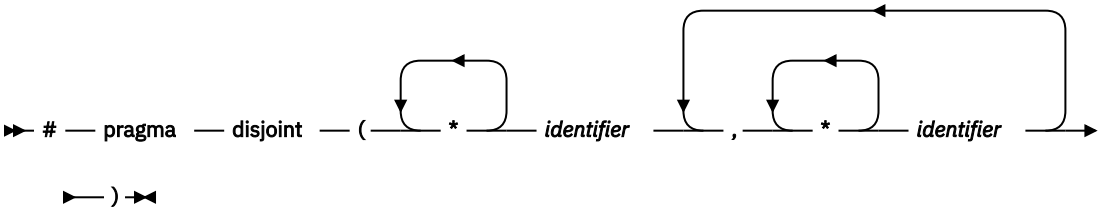
This directive is only needed when a handler has to be explicitly disabled before the end of a function. This is done since all enabled handlers are implicitly disabled at the end of the function in which they are enabled.

Notes on Usage

This pragma can only occur at a C language statement boundary and inside a function definition. The compiler issues an error message if the #pragma disable_handler is specified when no handler is currently enabled.

disjoint

disjoint syntax



Description

This directive informs the compiler that none of the identifiers listed shares the same physical storage, which provides more opportunity for optimizations. If any identifiers actually share physical storage, the pragma may cause the program to give incorrect results.

An identifier in the directive must be visible at the point in the program where the pragma appears. The identifiers in the disjoint name list cannot refer to any of the following items:

- a member of a structure, or union
- a structure, union, or enumeration tag
- an enumeration constant
- a typedef name
- a label

Example

```
int a, b, *ptr_a, *ptr_b;
#pragma disjoint(*ptr_a, b)  // *ptr_a never points to b
#pragma disjoint(*ptr_b, a)  // *ptr_b never points to a
one_function()
{
    b = 6;
    *ptr_a = 7;     // Assignment does not alter the value of b
    another_function(b);    // Argument "b" has the value 6
}
```

Because external pointer `ptr_a` does not share storage with and never points to the external variable b, the assignment of 7 to the object that `ptr_a` points to will not change the value of b. Likewise, external pointer `ptr_b` does not share storage with and never points to the external variable a. The compiler can assume that the argument of another_function has the value 6 and will not reload the variable from memory.

do_not_instantiate

C++

do_not_instantiate syntax

Description

Prevents the specified template declaration from being instantiated. You can use this pragma to suppress the implicit instantiation of a template for which a definition is supplied.

Parameters

template_class_name
The name of the template class that should not be instantiated.

Notes on Usage

If you are handling template instantiations manually (that is, compiler options `TEMPLATE(*NONE)` and `TMPLREG(*NONE)` are in effect), and the specified template instantiation exists in another compilation

unit, using `#pragma do_not_instantiate` ensures that you do not get multiple symbol definitions during the link step.

`#pragma do_not_instantiate` on a class template specialization is treated as an explicit instantiation declaration of the template. This pragma provides a subset of the functionality of the explicit instantiation declarations feature, which is introduced by the C++0x standard. It is provided for compatibility purposes only and is not recommended. New applications should use explicit instantiation declarations instead. See “LANGLVL” on page 80 and "Explicit instantiation(C++ only)" in *ILE C/C++ Language Reference*.

Examples

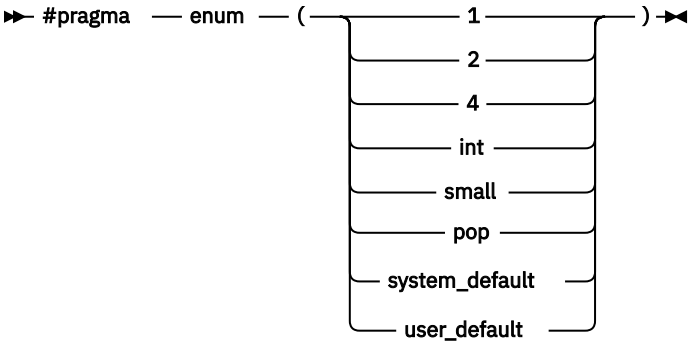
The following example shows the usage of the pragma:

```
#pragma do_not_instantiate Stack <int>
```

enum

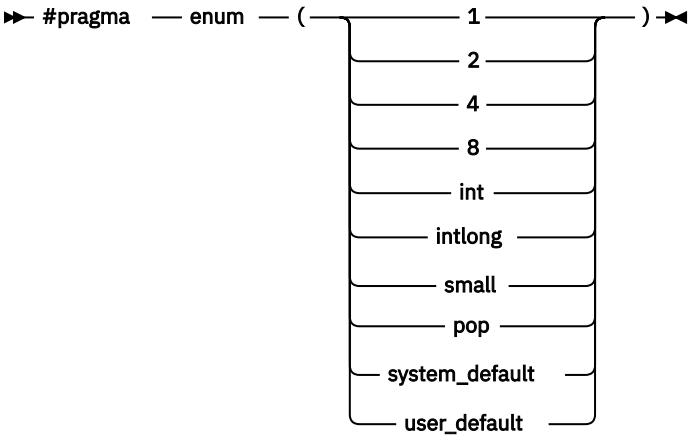
C

enum syntax



C++

enum syntax



Description

Specifies the number of bytes the compiler uses to represent enumerations. The pragma affects all subsequent enum definitions until the end of the compilation unit or until another #pragma enum directive is encountered. If more than one pragma is used, the most recently encountered pragma is in effect. This pragma overrides the ENUM compiler option, described on page “ENUM” on page 93.

Parameters

1, 2, 4, 8
Specifies that enumerations be stored in 1, 2, 4, or 8-byte containers. The sign of the container is determined by the range of values in the enumeration, but preference is given to signed when the range permits either. The pragma `enum(8)` directive is only available in C++.

int
Causes enumerations to be stored in the ANSI C or C++ Standard representation of an enumeration, which is 4-bytes signed. In C++ programs, the int container may become 4-bytes unsigned if a value in the enumeration exceeds 2^{31}-1, as per the ANSI C++ Standard.

intlong
Specifies that enumerations occupy 8 bytes of storage if the range of values in the enumeration exceeds the limit for int. If the range of values in the enumeration does not exceed the limit for int, the enumeration will occupy 4 bytes of storage and is represented as though enum(int) was specified. The pragma `enum(intlong)` directive is only available in C++

small
Causes subsequent enumerations to be placed into the smallest possible container, given the values in the enumeration. The sign of the container is determined by the range of values in the enumeration, but preference is given to unsigned when the range permits either.

pop
Selects the enumeration size previously in effect, and discards the current setting.

system_default
Selects the default enumeration size, which is the `small` option.

user_default
Selects the enumeration size specified by the `ENUM` compiler option.

The value ranges that can be accepted by the enum settings are shown below:

Table 2. Value Ranges Accepted by the enum Settings

Range of Element Values	Enum Options						
	small (default)	**1**	**2**	**4**	**8 (C++ only)**	**int**	**intlong (C++ only)**
0 .. 127	1 byte unsigned	1 byte signed	2 bytes signed	4 bytes signed	8 bytes signed	4 bytes signed	4 bytes signed
0 .. 255	1 byte unsigned	1 byte unsigned	2 bytes signed	4 bytes signed	8 bytes signed	4 bytes signed	4 bytes signed
-128 .. 127	1 byte signed	1 byte signed	2 bytes signed	4 bytes signed	8 bytes signed	4 bytes signed	4 bytes signed
0 .. 32767	2 bytes unsigned	ERROR	2 bytes signed	4 bytes signed	8 bytes signed	4 bytes signed	4 bytes signed
0 .. 65535	2 bytes unsigned	ERROR	2 bytes unsigned	4 bytes signed	8 bytes signed	4 bytes signed	4 bytes signed
-32768 .. 32767	2 bytes signed	ERROR	2 bytes signed	4 bytes signed	8 bytes signed	4 bytes signed	4 bytes signed

Table 2. Value Ranges Accepted by the enum Settings (continued)

Range of Element Values	Enum Options						
	small (default)	**1**	**2**	**4**	**8 (C++ only)**	**int**	**intlong (C++ only)**
0 .. 2147483647	4 bytes unsigned	ERROR	ERROR	4 bytes signed	8 bytes signed	4 bytes signed	4 bytes signed
0 .. 4294967295	4 bytes unsigned	ERROR	ERROR	4 bytes unsigned	8 bytes signed	C++ 4 bytes unsigned C ERROR	4 bytes unsigned
-2147483648 .. 2147483647	4 bytes signed	ERROR	ERROR	4 bytes signed	8 bytes signed	4 bytes signed	4 bytes signed
0 .. (2^{63} -1) (C++ only)	8 bytes unsigned	ERROR	ERROR	ERROR	8 bytes signed	ERROR	8 bytes signed
0.. 2^{64} (C++ only)	8 bytes unsigned	ERROR	ERROR	ERROR	8 bytes unsigned	ERROR	8 bytes unsigned
-2^{63} .. (2^{63}-1) (C++ only)	8 bytes signed	ERROR	ERROR	ERROR	8 bytes signed	ERROR	8 bytes signed

Examples

The examples below show various uses of the #pragma enum and compiler options:

1. You cannot change the storage allocation of an enum by using #pragma enum within the declaration of an enum. The following code segment generates a warning and the second occurrence of the enum option is ignored:

```
#pragma enum ( small )
   enum e_tag { a, b,
#pragma enum ( int ) /* error: cannot be within a declaration */
   c
  } e_var;

#pragma enum ( pop ) /* second pop isn't required */
```

2. The range of enum constants must fall within the range of either unsigned int or int (signed int) for the C compiler. The range of enum constants must fall within the range of either unsigned long long or long long (signed long long) for the C++ compiler. For example, the following code segments contain errors when the C compiler is used, but will compile successfully when the C++ compiler is used.:

```
#pragma enum ( small )
   enum e_tag { a=-1,
                b=2147483648   /* C compiler error: larger than maximum int */
              } e_var;
#pragma enum ( pop )

#pragma enum ( small )
   enum e_tag { a=0,
```

```
                b=4294967296 /* C compiler error: larger than maximum int */
              } e_var;
#pragma enum ( pop )
```

3. One use for the pop option is to pop the enumeration size set at the end of an include file that specifies an enumeration storage different from the default in the main file. For example, the following include file, small_enum.h, declares various minimum-sized enumerations, then resets the specification at the end of the include file to the last value on the option stack:

```
#ifndef small_enum_h
#define small_enum_h
/*
 * File small_enum.h
 * This enum must fit within an unsigned char type
*/
#pragma enum ( small )
   enum e_tag {a, b=255};
   enum e_tag u_char_e_var; /* occupies 1 byte of storage */

/* Pop the enumeration size to whatever it was before */
#pragma enum ( pop )
#endif
```

   The following source file, int_file.c, includes small_enum.h:

```
/*
 * File int_file.c
 * Defines 4 byte enums
*/
#pragma enum ( int )
   enum testing {ONE, TWO, THREE};
   enum testing test_enum;

/* various minimum-sized enums are declared */
#include "small_enum.h"

/* return to int-sized enums. small_enum.h has popped the enum size
*/
   enum sushi {CALIF_ROLL, SALMON_ROLL, TUNA, SQUID, UNI};
   enum sushi first_order = UNI;
```

   The enumerations test_enum and first_order both occupy 4 bytes of storage and are of type int. The variable u_char_e_var defined in small_enum.h occupies 1 byte of storage and is represented by an unsigned char data type.

4. If the code fragment below is compiled with the ENUM = *SMALL option:

```
enum e_tag {a, b, c} e_var;
```

   the range of enum constants is 0 through 2. This range falls within all of the ranges described in the table above. Based on priority, the compiler uses predefined type unsigned char.

5. If the code fragment below is compiled with the ENUM = *SMALL option:

```
enum e_tag {a=-129, b, c} e_var;
```

   the range of enum constants is -129 through -127. This range only falls within the ranges of short (signed short) and int (signed int). Because short (signed short) is smaller, it will be used to represent the enum.

6. If you compile a file myprogram.c using the command:

```
CRTBNDC MODULE(MYPROGRAM) SRCMBR(MYPROGRAM) ENUM(*SMALL)
```

   all enum variables within your source file will occupy the minimum amount of storage, unless #pragma enum directives override the ENUM option.

7. If you compile a file yourfile.c that contains the following lines:

```
enum testing {ONE, TWO, THREE};
enum testing test_enum;

#pragma enum ( small )
```

```
enum sushi {CALIF_ROLL, SALMON_ROLL, TUNA, SQUID, UNI};
enum sushi first_order = UNI;

#pragma enum ( int )
enum music {ROCK, JAZZ, NEW_WAVE, CLASSICAL};
enum music listening_type;
```

using the command:

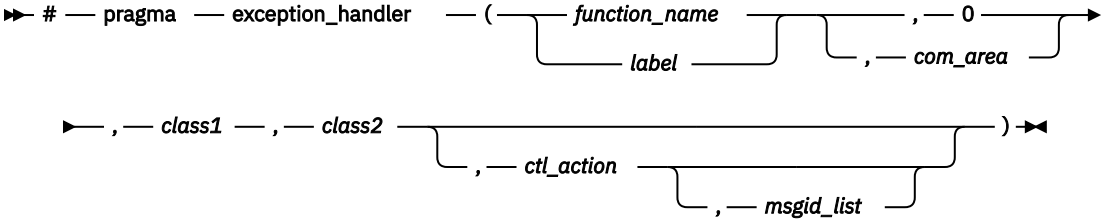
```
CRTBNDC MODULE(YOURFILE) SRCMBR(YOURFILE)
```

the enum variables test_enum and first_order will be minimum-sized (that is, each will only occupy 1 byte of storage). The other enum variable, listening_type, will be of type int and occupy 4 bytes of storage.

exception_handler

C C++

exception_handler syntax



Description

Enables a user-defined ILE exception handler at the point in the code where the #pragma exception_handler is located.

Any exception handlers enabled by #pragma exception_handler that are not disabled using #pragma disable_handler are implicitly disabled at the end of the function in which they are enabled.

Parameters

function
Specifies the name of the function to be used as a user-defined ILE exception handler.

label
Specifies the name of the label to be used as a user-defined ILE exception handler. The label must be defined within the function where the #pragma exception_handler is enabled. When the handler gets control, the exception is implicitly handled and control resumes at the label defined by the handler in the invocation containing the #pragma exception_handler directive. The call stack is canceled from the newest call to, but not including, the call containing the #pragma exception_handler directive. The label can be placed anywhere in the statement part of the function definition, regardless of the position of the #pragma exception_handler.

com_area
Used for the communications area. If no *com_area* should be specified, zero is used as the second parameter of the directive. If a *com_area* is specified on the directive, it must be a variable of one of the following data types: integral, float, double, struct, union, array, enum, pointer, or packed decimal. The com_area should be declared with the volatile qualifier. It cannot be a member of a structure or a union.

class1, class2
Specifies the first four bytes and the last four bytes of the exception mask. The <except.h> header file describes the values that you can use for the class masks. It also contains macro definitions for these values. *class1* and *class2* must evaluate to integer constant expressions after any necessary macro expansions. You can monitor for the valid *class2* values of:

- _C2_MH_ESCAPE

- _C2_MH_STATUS
- _C2_MH_NOTIFY, and
- _C2_FUNCTION_CHECK.

ctl_action

Specifies an integer constant to indicate what action should take place for this exception handler. If handler is a function, the default value is _CTLA_INVOKE. If handler is a label, the default value is _CTLA_HANDLE. This parameter is optional.

The following are valid exception control actions that are defined in the <except.h> header file:

#define name	**Defined value and action**
_CTLA_INVOKE	Defined to 1. This control action will cause the function named on the directive to be invoked and will not handle the exception. If the exception is not explicitly handled, processing will continue. This is valid for functions only.
_CTLA_HANDLE	Defined to 2. The exception is handled and messages are logged before calling the handler. The exception will no longer be active when the handler gets control. Exception processing ends when the exception handler returns. This is valid for functions and labels.
_CTLA_HANDLE_NO_MSG	Defined to 3. The exception is handled but messages are not logged before calling the handler. The exception will no longer be active when the handler gets control. Exception messages are not logged. Msg_Ref_Key in the typedef _INTRPT_Hndlr_Parms_T is set to zero. Exception processing ends when the exception handler returns. This is valid for functions and labels.
_CTLA_IGNORE	Defined to 131. The exception is handled and messages are logged. Control is not passed to the handler function named on the directive and exception will no longer be active. Execution resumes at the instruction immediately following the instruction that caused the exception. This is valid for functions only.
_CTLA_IGNORE_NO_MSG	Defined to 132. The exception is handled and messages are not logged. Control is not passed to the handler function named on the directive and exception will no longer be active. Execution resumes at the instruction immediately following the instruction that caused the exception. This is valid for functions only.

msgid_list

Specifies an optional string literal that contains the list of message identifiers. The exception handler will take effect only when an exception occurs whose identifiers match one of the identifiers on the list of message identifiers. The list is a series of 7-character message identifiers where the first three characters are the message prefix and the last four are the message number. Each message identifier is separated by one or more spaces or commas. This parameter is optional, but if it is specified, *ctl_action* must also be specified.

For the exception handler to get control, the selection criteria for *class1* and *class2* must be satisfied. If the *msgid_list* is specified, the exception must also match at least one of the message identifiers in the list, based on the following criteria:

- The message identifier matches the exception exactly.
- A message identifier, whose two rightmost characters are 00, will match any exception identifier that has the same five leftmost characters. For example, a message identifier of CPF5100 will match any exceptions whose message identifier begins with CPF51.

- A message identifier, whose four rightmost characters are 0000, will match any exception identifier that has the same prefix. For example, a message identifier of CPF0000 will match any exception whose message identifier has the prefix CPF (CPF0000 to CPF9999).
- If *msgid_list* is specified, but the exception that is generated is not one specified in the list, the exception handler will not get control.

Notes on Usage

The handler function can take only 16-byte pointers as parameters.

The macro _C1_ALL, defined in the <except.h> header file, can be used as the equivalent of all the valid *class1* exception masks. The macro _C2_ALL, defined in the <except.h> header file, can be used as the equivalent of all four of the valid *class2* exception masks.

You can use the binary OR operator to monitor for different types of messages. For example,

```
#pragma exception_handler(myhandler, my_comarea, 0, _C2_MH_ESCAPE | \
                _C2_MH_STATUS | _C2_MH_NOTIFY, _CTLA_IGNORE, "MCH0000")
```

will set up an exception monitor for three of the four *class2* exception classes that can be monitored.

The compiler issues an error message if any of the following occurs:

- The directive occurs outside a C function body or inside a C statement.
- The handler that is named is not a declared function or a defined label.
- The *com_area* variable has not been declared or does not have a valid object type.
- Either of the exception class masks is not a valid integral constant
- The *ctl_action* is one of the disallowed values when the handler that is specified is a label (_CTLA_INVOKE, _CTLA_IGNORE, _CTLA_IGNORE_NO_MSG).
- The *msgid_list* is specified, but the *ctl_action* is not.
- A message in the *msgid_list* is not valid. Message prefixes that are not in uppercase are not considered valid.
- The messages in the string are not separated by a blank or comma.
- The string is not enclosed in “ ” or is longer than 4 KB.
- The handler function specified is defined with argument optimization (#pragma argopt).

If a label is used as a user-defined exception handler, some of the code between the exception_handler pragma and the disable_handler pragma could be skipped if an exception occurs. Therefore, any declaration or statement in the exception range may be skipped causing variables to not be initialized or variable length arrays to not have storage allocated (since storage is allocated for variable length arrays at the time of their declaration). In the following example

1. The assignment to `z` may cause an exception if `i` is zero. If so, control branches to LABEL.
2. If an exception occurs, `ptr` will not set to `&z` and `vla` will not have its storage allocated.
3. The use of `*ptr` and `vla[0]` may be illegal if `ptr` is not initialized and the storage for `vla` is not allocated.

```
void func(unsigned int i)
{
  unsigned int *ptr = NULL;
  unsigned int z;
#pragma exception_handler(LABEL,0, _C1_ALL, _C2_ALL)
  z = 45/i;    // 1
  ptr = &z;    // 2
  unsigned int vla[z];
#pragma disable_handler
LABEL:
  vla[0] = *ptr;   // 3
  return;
}
```

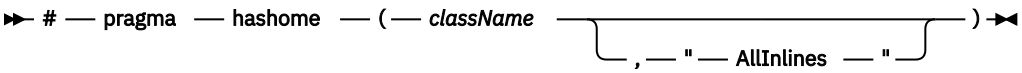
In addition, because variable length arrays declare their storage at run time and that may cause a storage allocation exception, it is recommended to have an exception handler enabled for variable length array declarations.

See the *ILE C/C++ Programmer's Guide* for examples and more information about using the #pragma exception_handler directive.

hashome

C++

hashome syntax



Description

Informs the compiler that the specified class has a home module that will be specified by #pragma ishome. This class's virtual function table, along with certain inline functions, will not be generated as static. Instead, they will be referenced as externals in the compilation unit of the class in which #pragma ishome was specified.

Parameters

className
Specifies the name of a class that requires the above mentioned external referencing. *className* must be a class and it must be defined.

AllInlines
specifies that all inline functions from within *className* should be referenced as being external. This argument is case insensitive.

A warning will be produced if there is a #pragma ishome without a matching #pragma hashome.

See also "ishome" on page 41.

implementation

C++

implementation syntax

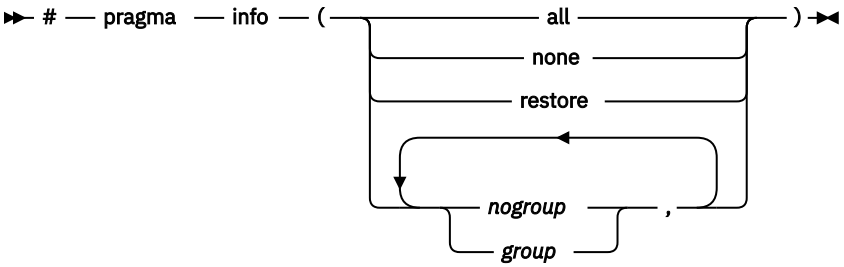
Description

The #pragma implementation directive tells the compiler the name of the file containing the function-template definitions that correspond to the template declarations in the include file which contains the pragma. This pragma can appear anywhere that a declaration is allowed. It is used when organizing your program for the efficient or automatic generation of template functions.

info

C++

info syntax



Description

This pragma can be used to control which diagnostic messages are produced by the compiler.

Parameters

all
Generates all diagnostic messages while this pragma is in effect.

none
Turns off all diagnostic messages while this pragma is in effect.

restore
Restores the previous setting of `pragma info`.

nogroup
Suppresses all diagnostic messages associated with a specified diagnostic group. To turn off a specific group of messages, prepend the group name with "no". For example, `nogen` will suppress CHECKOUT messages. Valid group names are listed below.

group
Generates all diagnostic messages associated with the specified diagnostic group. Valid group names are:

lan
Display information about the effects of the language level

gnr
Generate messages if the compiler creates temporary variables

cls
Display information about class use

eff
Warn about statements with no effect

cnd
Warn about possible redundancies or problems in conditional expressions

rea
Warn about unreachable statements

par
List the function parameters that are not used

por
List the non-portable usage of the C/C++ language

trd
Warn about the possible truncation or loss of data

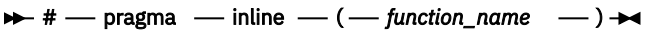
use
Check for unused auto or static variables

gen
List the general CHECKOUT messages

inline

C

inline syntax



Description

The #pragma inline directive specifies that *function_name* is to be inlined. The pragma can appear anywhere in the source, but must be at file scope. The pragma has no effect if the INLINE(*ON) compiler option parameter is not specified. If **#pragma inline** is specified for a function, the inliner will force the function specified to be inlined on every call. The function will be inlined in both selective (*NOAUTO) and automatic (*AUTO) INLINE mode.

Inlining replaces function calls with the actual code of the function. It reduces function call overhead, and exposes more code to the optimizer, allowing more opportunities for optimization.

Notes on Usage

- Inlining takes place only if compiler optimization is set to level 30 or higher.
- Directly recursive functions will not be inlined. Indirectly recursive functions will be inlined until direct recursion is encountered.
- Functions calls with variable argument lists will not be inlined if arguments are encountered in the variable portion of the argument list.
- If a function is called through a function pointer, then inlining will not occur.
- The pragma inline directive will be ignored if *function_name* is not defined in the same compilation unit that contains the pragma.
- A function's definition will be discarded if all of the following are true:
  - The function is static.
  - The function has not had its address taken.
  - The function has been inlined everywhere it is called.

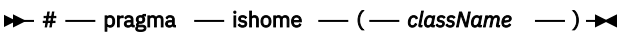
  This action can decrease the size of the module and program object where the function is used.

See "Function Call Performance" in the *ILE C/C++ Programmer's Guide* for more information about function inlining.

ishome

C++

ishome syntax



Description

Informs the compiler that the specified class's home module is the current compilation unit. The home module is where items, such as the virtual function table, are stored. If an item is referenced from outside of the compilation unit, it will not be generated outside its home. The advantage of this is the minimization of code.

Parameters

className
Specifies the literal name of the class whose home will be the current compilation unit.

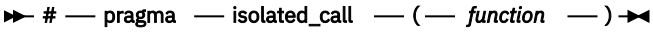
A warning will be produced if there is a #pragma ishome without a matching #pragma hashome.

See also “hashome” on page 39.

isolated_call

C++

isolated_call syntax



Description

Lists a function that does not have or rely on side effects, other than those side effects implied by its parameters.

Parameters

function
Specifies a primary expression that can be an identifier, operator function, conversion function, or qualified name. An identifier must be of type function or a typedef of function. If the name refers to an overloaded function, all variants of that function are marked as isolated calls.

Notes on Usage

The pragma informs the compiler that the function listed does not have or rely on side effects, other than those side effects implied by its parameters. Functions are considered to have or rely on side effects if they:

- Access a volatile object
- Modify an external object
- Modify a static object
- Modify a file
- Access a file that is modified by another process or thread
- Allocate a dynamic object, unless it is released before returning
- Release a dynamic object, unless it was allocated during the same invocation
- Change system state, such as rounding mode or exception handling
- Call a function that does any of the previous

Essentially, any change in the state of the runtime environment is considered a side effect. Modifying function arguments passed by pointer or by reference is the only side effect that is allowed. Functions with other side effects can give incorrect results when listed in #pragma isolated_call directives.

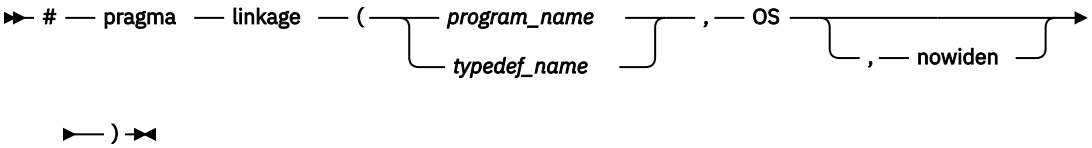
Marking a function as isolated_call indicates to the optimizer that external and static variables cannot be changed by the called function and that pessimistic references to storage can be deleted from the calling function where appropriate. Instructions can be reordered with more freedom, resulting in fewer pipeline delays, and faster execution in the processor. Multiple calls to the same function with identical parameters can be combined, calls can be deleted if their results are not needed, and the order of calls can be changed.

The function specified is permitted to examine non-volatile external objects and return a result that depends on the non-volatile state of the runtime environment. The function can also modify the storage pointed to by any pointer arguments passed to the function, that is, calls by reference. Do not specify a function that calls itself or relies on local static storage. Listing such functions in the #pragma isolated_call directive can give unpredictable results.

linkage

C

linkage syntax



Description

Identifies a given function or function typedef as being an external program subject to IBM i parameter passing conventions.

This pragma allows calls only to external programs. For information about making calls to bound procedures, see #pragma “argument” on page 23 .

Parameter

program_name
Specifies the name of an external program. The external name must be specified in uppercase characters and be no longer than 10 characters in length, unless the #pragma map directive is specified to meet IBM i program naming conventions. However, if the name specified in #pragma map is too long, it will be truncated to 255 characters during #pragma linkage processing.

typedef_name
Specifies a typedef affected by this pragma.

OS
Specifies that the external program is called using IBM i calling conventions.

nowiden
If specified, arguments are not widened before they are copied and passed.

Notes on Usage

This pragma lets an IBM i program call an external program. The external program can be written in any language.

The pragma can be applied to functions, function types, and function pointer types. If it is applied to a function typedef, the effect of the pragma also applies to all functions and new typedefs declared using that original typedef.

This directive can appear either before or after the program name (or type) is declared. However, the program cannot have been called, nor a type been used in a declaration, before the pragma directive.

The function or function pointer can only return either an `int` or a `void`.

Arguments on the call are passed according to the following IBM i argument-passing conventions:

- Non-address arguments are copied to temporary locations, widened (unless `nowiden` has been specified) and the address of the copy is passed to the called program.
- Address arguments are passed directly to the called program.

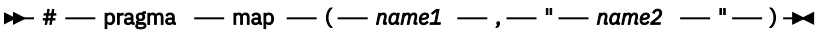
The compiler issues a warning message and ignores the #pragma linkage directive if:

- The program is declared with a return type other than `int` or `void`.
- The function contains more than 256 parameters.
- Another pragma linkage directive has already been specified for the function or function type.
- The function has been defined in the current compilation unit.
- The specified function has already been called, or the type already used in a declaration.
- #pragma argopt or #pragma argument has already been specified for the named function or type.
- The object named in the pragma directive is not a function or function type.
- The name of the object specified in the pragma directive must not exceed 10 characters, or the name will be truncated.

map

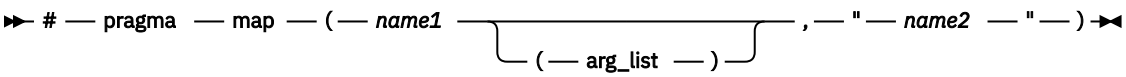
C

map syntax



C++

map syntax



Description

Converts all references to an identifier to another, externally defined identifier.

Parameters

name1

The name used in the source code. For C, *name1* can represent a data object or function with external linkage. For C++, *name1* can represent a data object, a non-overloaded or overloaded function, or overloaded operator, with external linkage. If the name to be mapped is not in the global namespace, it must be fully qualified.

name1 should be declared in the same compilation unit in which it is referenced, but should not be defined in any other compilation unit. *name1* must not be used in another `#pragma map` directive anywhere in the program.

arg_list

The list of arguments for the overloaded function or operator function designated by *name1*. If *name1* designates an overloaded function, the function must be parenthesized and must include its argument list if it exists. If *name1* designates a non-overloaded function, only *name1* is required, and the parentheses and argument list are optional.

name2

The name that appears in the object code. If *name2* exceeds 65535 bytes in length, a message is issued and the pragma is ignored.

name2 can be declared or defined in the same compilation unit in which *name1* is referenced. *name2* must not be the same as that used in another `#pragma map` directive in the same compilation unit.

Notes on Usage

The `#pragma map` directive can appear anywhere in the program.

In order for a function to be actually mapped, the map target function (*name2*) must have a definition available at link time.

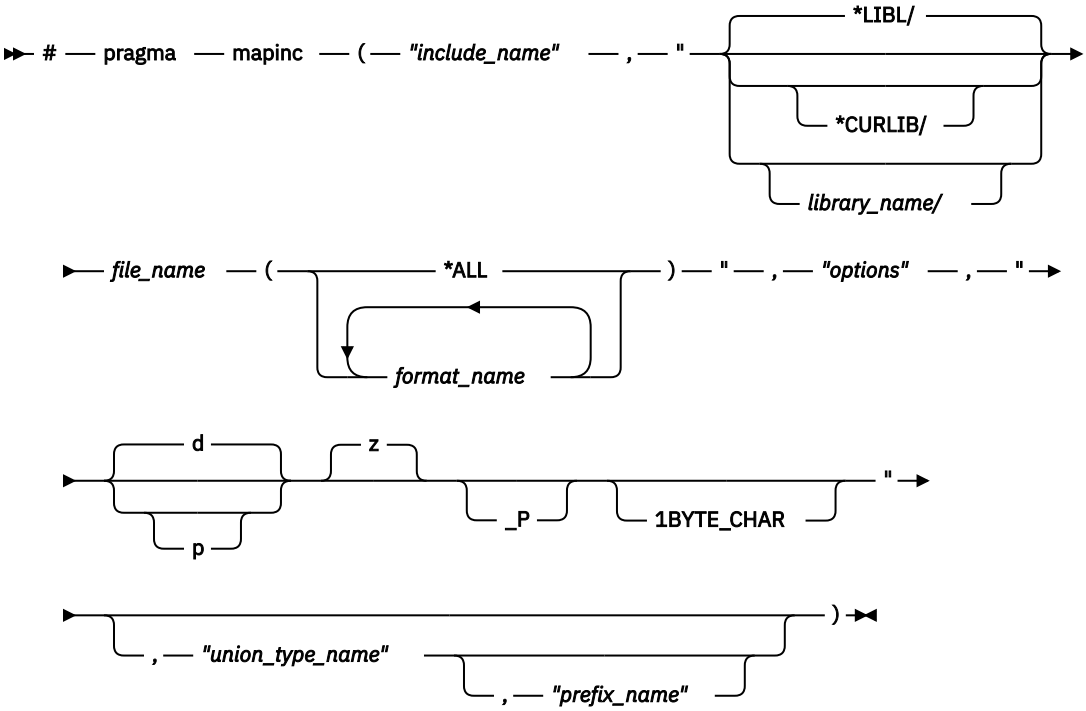
If *name2* specifies a function name that uses C++ linkage, then *name2* must be specified using its mangled name. For example,

```
int foo(int, int);
#pragma map(foo, "bar__FiT1")

int main( )
{
  return foo(4,5);
}

int bar(int a, int b)
{
  return a+b;
}
```

mapinc

C

mapinc syntax



Description

Indicates that data description specifications (DDS) are to be included in a module. The directive identifies the file and DDS record formats, and provides information about the fields to be included. This pragma, along with its associated include directive, causes the compiler to automatically generate typedefs from the record formats that are specified in the external file descriptions.

Parameters

include_name
This is the name that you refer to on the #include directive in the source program.

library_name
This is the name of the library that contains the externally described file

file_name
This is the name of the externally described file.

format_name
This is a required parameter which indicates the DDS record format that is to be included in your program. You can include more than one record format (format1 format2), or all the formats in a file (*ALL).

options
The possible *options* are:

input
Fields declared as either INPUT or BOTH in the DDS are included in the typedef structure. Response indicators are included in the input structure when the keyword INDARA is not specified in the external file description (DDS source) for device files.

output
Fields declared as either OUTPUT or BOTH in DDS are included in the typedef structure. Option indicators are included in the output structure when the keyword INDARA is not specified in the external file description (DDS source) for device files.

both
Fields declared as INPUT, OUTPUT, or BOTH in DDS are included in the typedef structure. Option and response indicators are included in both structures when the keyword INDARA is not specified in the external file description (DDS source) for device files.

key
Fields that are declared as keys in the external file description are included. This option is only valid for database files and DDM files.

indicators
A separate 99-byte structure for indicators is created when the indicator option is specified. This option is only valid for device files.

lname
This option allows the use of file names of up to 128 characters in length. If the file name has more than 10 characters then the name will be converted to an associated short name. The short name will be used to extract the external file definition. When the file has a short name of 10 characters or less the name is not converted to an associated short name. Record field names up to 30 characters in length will be generated in the typedefs by the compiler.

lvlchk
A typedef of an array of struct is generated (type name _LVLCHK_T) for the level check information. A pointer to an object of type _LVLCHK_T is also generated and is initialized with the level check information (format name and level identifier).

nullflds
If there is at least one null-capable field in the record format of the DDS, a null map typedef is generated containing a character field for every field in the format. With this typedef, the user can specify which fields are to be considered null (set value of each null field to 1, otherwise set to zero). Also, if the key option is used along with option nullflds, and there is at least one null-capable key field in the format, an additional typedef is generated containing a character field for every key field in the format.

For physical and logical files you can specify `input`, `both`, `key`, `lvlchk`, and `nullflds`. For device files you can specify `input`, `output`, `both`, `indicator`, and `lvlchk`.

The data type can be one or more of the following and must be separated by spaces.

d
Packed decimal data type.

p
Packed fields from DDS are declared as character fields.

z
Zoned fields from DDS are declared as character fields. This is the default because the compiler does not have a zoned data type.

_P
Packed structure is generated.

1BYTE_CHAR
Generates a single byte character field for one byte characters that are defined in DDS.

" "
Default values of d and z are used.

union_type_name
A union definition of the included type definitions is created with the name union_type_name_t. This parameter is optional.

prefix_name
Specifies the first part of the generated typedef structure name. If the prefix is not specified, the library and file_name are used.

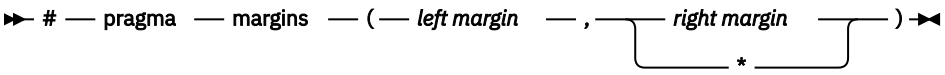
Notes on Usage

See *Using Externally Described Files in a Program* in the *ILE C/C++ Programmer's Guide* for more information about using the #pragma mapinc directive with externally described files.

margins

C

margins syntax



Description

Specifies the left margin to be used as the first column, and the right margin to be used as the last column, when scanning the records of the source member where the #pragma directive occurs.

The margin setting applies only to the source member in which it is located and has no effect on any source members named on include directives in the member.

Parameters

left margin
Must be a number greater than zero but less than 32 754. The *left margin* should be less than the *right margin*.

right margin
Must be a number greater than zero but less than 32 754, or an asterisk (*). The *right margin* should be greater than the *left margin*. The compiler scans between the left margin and the right margin. The compiler scans from the left margin specified to the end of the input record, if an asterisk is specified as the value of *right margin*.

Notes on Usage

The #pragma margins directive takes effect on the line following the directive and remains in effect until another #pragma margins or nomargins directive is encountered or the end of the source member is reached.

The #pragma margins and #pragma sequence directives can be used together. If these two #pragma directives reserve the same columns, the #pragma sequence directive has priority, and the columns are reserved for sequence numbers.

For example, if the #pragma margins directive specifies margins of 1 and 20, and the #pragma sequence directive specifies columns 15 to 25 for sequence numbers, the margins in effect are 1 and 14, and the columns reserved for sequence numbers are 15 to 25.

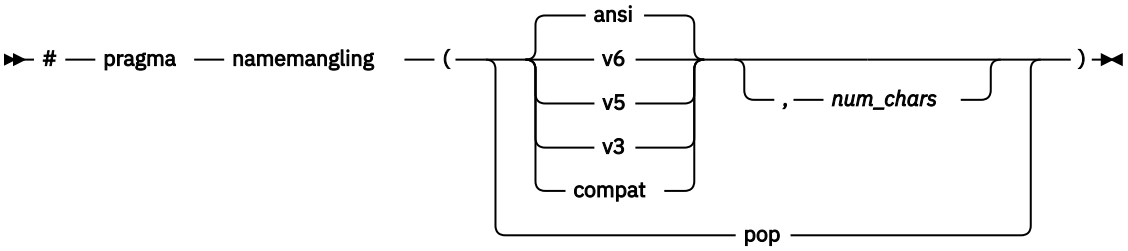
If the margins specified are not in the supported range or the margins contain non-numeric values, a warning message is issued during compilation and the directive is ignored.

See also pragmas “nomargins” on page 50 and “sequence” on page 58.

namemangling

C++

namemangling syntax



Description

Chooses the name mangling scheme for external symbol names generated from C++ source code. The option and pragma are provided to ensure binary compatibility with link modules created with previous versions of the compiler. If you do not need to ensure backwards compatibility, it is recommended that you do not change the default setting of this option.

Parameters

ansi
The name mangling scheme fully supports the most recent language features of Standard C++, including function template overloading. **ansi** is the default.

v6
The name mangling scheme is the same as used in the V5R3M0, V5R4M0, and V6R1M0 versions of the compiler.

v5
The name mangling scheme is the same as used in the V5R1M0 and V5R2M0 versions of the compiler.

v3
The name mangling scheme is the same as in versions of the compiler before V5R1M0.

compat
This option is the same as **v3**, described previously.

num_chars
Specifies the maximum number of allowable characters in the mangled names. If you do not specify this suboption, the default maximum is 64000 characters for all settings except **v3** and **compat**, for which the default maximum is 255 characters.

pop
Discards the current pragma setting and reverts to the setting specified by the previous pragma directive. If no previous pragma was specified, the default setting of **ansi** is used.

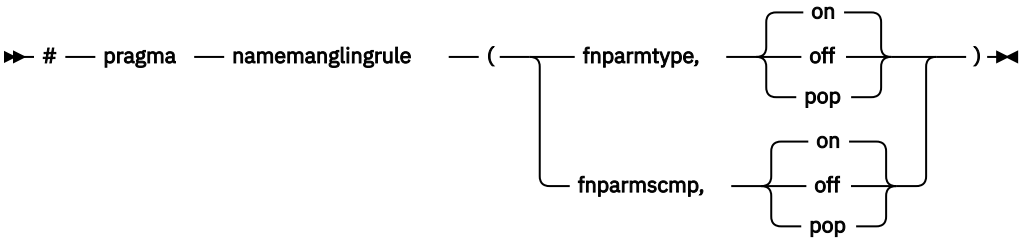
Note on Usage

The **#pragma namemangling** directive is not supported if the RTBND(*LLP64) compile option is used.

namemanglingrule

C++

namemanglingrule syntax



Description

Provides fine-grained control over the name mangling scheme in effect for selected portions of source code, specifically with respect to the mangling of cv-qualifiers in function parameters.

The `#pragma namemanglingrule` directive allows you to control whether top-level cv-qualifiers are mangled in function parameters or whether intermediate-level cv-qualifiers are to be considered when the compiler compares repeated function parameters for equivalence.

Defaults

`fnparmtype, on` when `#pragma namemangling(ansi|v6)` is in effect. Otherwise, the default is `fnparmtype, off`.

`fnparmscmp, on` when `#pragma namemangling(ansi)` is in effect. Otherwise, the default is `fnparmscmp, off`.

Parameters

fnparmtype, on
Top-level cv-qualifiers are not encoded in the mangled name of a function parameter. Also, top-level cv-qualifiers are ignored when repeated function parameters are compared for equivalence; function parameters that differ only by the use of a top-level cv-qualifier are considered equivalent and are mangled according to the compressed encoding scheme. This setting is compatible with ILE C++ V5R3M0 and later releases.

fnparmtype, off
Top-level cv-qualifiers are encoded in the mangled name of a function parameter. Also, repeated function parameters that differ by the use of cv-qualifiers are not considered equivalent and are mangled as separate parameters. This setting is compatible with ILE C++ V5R2M0 and earlier releases.

fnparmtype, pop
Reverts to the previous `fnparmtype` setting in effect. If no previous settings are in effect, the default `fnparmtype` setting is used.

fnparmscmp, on
Intermediate-level cv-qualifiers are considered when repeated function parameters are compared for equivalence; repeated function parameters that differ by the use of intermediate-level cv-qualifiers are mangled as separate parameters. This setting is compatible with ILE C++ V7R1M0 and later releases.

fnparmscmp, off
Intermediate-level cv-qualifiers are ignored when repeated function parameters are compared for equivalence; function parameters that differ only by the use of an intermediate-level cv-qualifier are considered equivalent and are mangled according to the compressed encoding scheme. This setting is compatible with ILE C++ V6R1M0 and earlier releases.

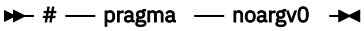
fnparmscmp, pop
Reverts to the previous `fnparmscmp` setting in effect. If no previous settings are in effect, the default `fnparmscmp` setting is used.

Notes on Usage

1. `#pragma namemanglingrule` is allowed in global, class, and function scopes. It has no effect on a block scope function declaration with external linkage.
2. Different pragma settings can be specified in front of function declarations and definitions. If `#pragma namemanglingrule` settings in subsequent declarations and definitions conflict, the compiler ignores those settings and issues a warning message.
3. The `#pragma namemanglingrule` directive is not supported if the RTBND(*LLP64) compile option is used.

noargv0

noargv0 syntax



Description

Specifies that the source program does not make use of argv[0]. This pragma can improve performance of applications that have a large number of small C programs, or a small program that is called many times.

Notes on Usage

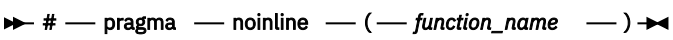
The #pragma noargv0 must appear in the compilation unit where the `main()` function is defined, otherwise it is ignored.

argv[0] will be NULL when the noargv0 pragma directive is in effect. Other arguments in the argument vector will not be affected by this directive. If the #pragma noargv0 directive is not specified, argv[0] will contain the name of the program that is currently running.

noinline (function)

C

noinline syntax



Description

Specifies that a function will not be inlined. The settings on the INLINE compiler option will be ignored for this function_name.

Notes on Usage

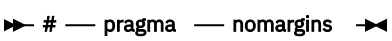
The first pragma specified will be the one that is used. If #pragma inline is specified for a function after #pragma noinline has been specified for it, a warning will be issued to indicate that #pragma noinline has already been specified for that function.

The #pragma noinline directive can only occur at file scope. The pragma will be ignored, and a warning is issued if it is not found at file scope.

nomargins

C

nomargins syntax



Description

Specifies that the entire input record is to be scanned for input.

Notes on Usage

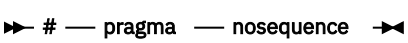
The #pragma nomargins directive takes effect on the line following the directive and remains in effect until a #pragma margins directive is encountered or the end of the source member is reached.

See also pragma “margins” on page 47.

nosequence

C

nosequence syntax



Description

Specifies that the input record does not contain sequence numbers.

Notes on Usage

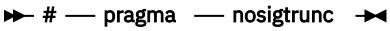
The #pragma nosequence directive takes effect on the line following the directive and remains in effect until a #pragma sequence directive is encountered or the end of the source member is reached.

See also pragma “sequence” on page 58.

nosigtrunc

C

nosigtrunc syntax



Description

Specifies that no exception is generated at runtime when overflow occurs with packed decimals in arithmetic operations, assignments, casting, initialization, or function calls. This directive suppresses the signal that is raised in packed decimal overflow. The #pragma nosigtrunc directive can only occur at filescope. A warning message will be issued if the #pragma nosigtrunc directive is encountered at function, block or function prototype scope, and the directive will be ignored.

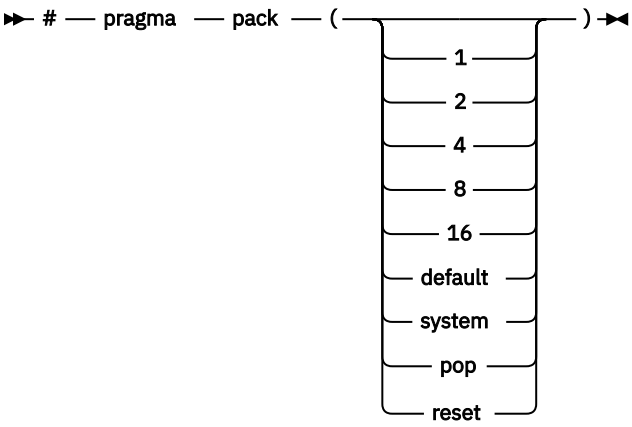
Notes on Usage

This #pragma directive has file scope and must be placed outside a function definition; otherwise it is ignored. A warning message may still be issued during compilation for some packed decimal operations if overflow is likely to occur. See the *ILE C/C++ Programmer's Guide* for more information about packed decimal errors.

pack

C C++

pack syntax



Description

The #pragma pack directive specifies the alignment rules to use for the members of the structure, union, or (C++ only) class that follows it. In C++, packing is performed on *declarations* or types. This is different from C, where packing is also performed on *definitions*.

You can also use the PACKSTRUCT option with the compiler commands to cause packing to be performed along specified boundaries. See "PACKSTRUCT" on page 92 for more information.

Parameters

1, 2, 4, 8, 16
Structures and unions are packed along the specified byte boundaries.

default
Selects the alignment rules specified by compiler option PACKSTRUCT.

system
Selects the default IBM i alignment rules.

pop, reset
Selects the alignment rules previously in effect, and discards the current rules. This is the same as specifying `#pragma pack ( )`.

In the examples that follow, the words *struct* or *union* can be used in place of *class*.

The #pragma pack settings are stack based. All pack values are pushed onto a stack as the user's source code is parsed. The value on the top of that stack is the current packing value. When a #pragma pack (reset), #pragma pack(pop), or #pragma pack() directive is given, the top of the stack is popped and the next element in the stack becomes the new packing value. If the stack is empty, the value of the PACKSTRUCT compiler option, if specified, is used. If not specified, the default setting of NATURAL alignment is used.

The setting of the PACKSTRUCT compiler option is overridden by the #pragma pack directive, but always remains on the bottom of the stack. The keyword _Packed has the highest precedence with respect to packing options, and cannot be overridden by the #pragma pack directive or the PACKSTRUCT compiler option.

By default, all members use their natural alignment. Members cannot be aligned on values greater than their natural alignment. Char types can only be aligned along 1-byte boundaries. Short types can only be aligned along 1 or 2-byte boundaries, and int types can be aligned along on 1, 2, or 4-byte boundaries.

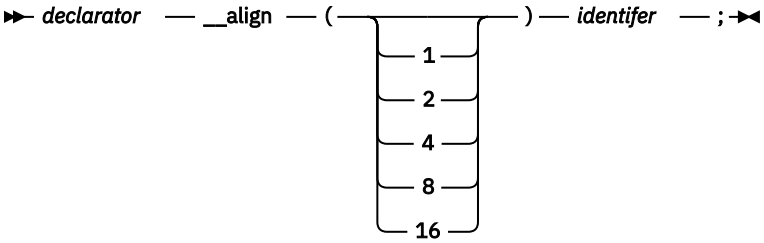
All 16-byte pointers will be aligned on a 16-byte boundary. _Packed, PACKSTRUCT, and #pragma pack cannot alter this. 8-byte teraspace pointers may have any alignment, although 8-byte alignment is preferred.

Related Operators and Specifiers

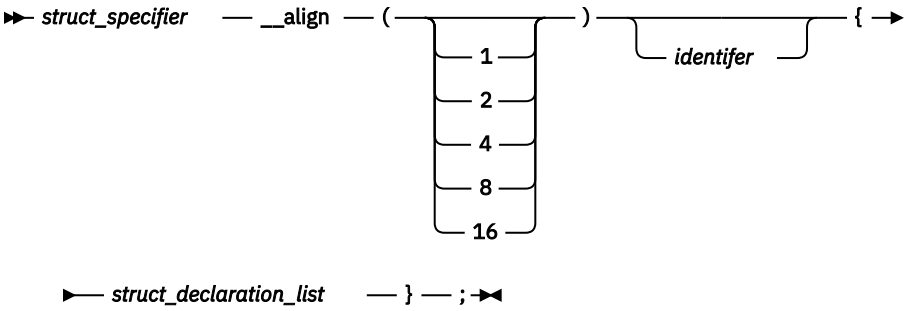
__align Specifier

The __align specifier lets you specify the alignment of a Data Item or a ILE C/C++ aggregate (such as a struct or union for ILE C, as well as classes for ILE C++). However, __align *does not* affect the alignment of members within an aggregate, only the alignment of the aggregate as a whole. Also, because of restrictions for certain members of an aggregate, such as 16-byte pointers, the alignment of an aggregate is not guaranteed to be aligned in memory on the boundary specified by __align. For example, an aggregate that has a 16-byte pointer as its only member cannot have any other alignment other than 16-byte alignment because all 16-byte pointers must be aligned on the 16-byte boundary.

__align syntax



struct_specifier syntax



You can also use the __align specifier to explicitly specify alignment when declaring or defining data items, as shown in some of the examples that follow.

The __align specifier:

- Can only be used with declarations of first-level variables and aggregate definitions. It ignores parameters and automatics.
- Cannot be used on individual elements within an aggregate definition, but it can be used on an aggregate definition nested within another aggregate definition.
- Cannot be used in the following situations:
  - Individual elements within an aggregate definition.
  - Variables declared with incomplete type.
  - Aggregates declared without definition.
  - Individual elements of an array.
  - Other types of declarations or definitions, such as function and enum.
  - Where the size of variable alignment is smaller than the size of type alignment.

_Packed Specifier

_Packed can be associated with struct, union, and in C++, class definitions. In C++, _Packed must be specified on a typedef. It has the same effect as #pragma pack(1). The following code shows examples of valid and invalid usages of _Packed. In these examples, the keywords *struct*, *union*, and *class* can be used interchangeably.

```
typedef _Packed class SomeClass { /* ... */ } MyClass;  // OK
typedef _Packed union AnotherClass {} PUnion;           // OK
typedef _Packed struct {} PAnonStruct;                  // Invalid, struct must be named
Class Stack { /* ... */ };
_Packed Stack someObject;                               // Invalid, specifier _Packed must be
                                                        // associated with a typedef in C++
_Packed struct SomeStruct { };                          // OK for C, invalid for C++
_Packed union SomeUnion { };                            // OK for C, invalid for C++
```

__alignof Operator

```
unary-expression:
    __alignof unary-expression
    __alignof ( type-name )
```

The __alignof operator returns the alignment of its operand, which may be an expression or the parenthesized name of a type. The alignment of the operand is determined according to IBM i alignment rules. However, it should not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member. The type of the result of this operator is size_t.

Examples

In the examples that follow, the words union and class might be used in place of the word struct.

1. **Popping the #pragma pack Stack**

   Specifying #pragma pack (pop), #pragma pack (reset), or #pragma pack() pops the stack by one and resets the alignment requirements to the state that was active before the previous #pragma pack was seen. For example,

   ```
   // Default alignment requirements used
   .
   .
   #pragma pack (4)
    struct A {  };
   #pragma pack (2)
    struct B {  };
    struct C {  };
   #pragma pack (reset)
    struct D {  };
   #pragma pack ()
    struct E {  };
   ```

```
    #pragma pack (pop)
     struct F {  };
```

When `struct A` is mapped, its members are aligned according to #pragma pack(4). When `struct B` and `struct C` are mapped, their members are aligned according to pragma pack(2).

The #pragma pack (reset) pops the alignment requirements specified by #pragma pack(2) and resets the alignment requirements as specified by #pragma pack(4).

When `struct D` is mapped, its members are aligned according to pragma pack(4). The #pragma pack () pops the alignment requirements specified by #pragma pack(4) and resets the alignment requirements to the default values used at the beginning of the file.

When `struct E` is mapped, its members are aligned as specified by the default alignment requirements (specified on the command line) active at the beginning of the file.

The #pragma pack (pop) has the same affect as the previous #pragma pack directives in that it pops the top value from the pack stack. However, the default pack value, as specified in the PACKSTRUCT compiler option, cannot be removed from the pack stack. That default value is used to align struct F.

2. **__align & #pragma pack**

```
__align(16) struct S {int i;};          /* sizeof(struct S) == 16   */
struct S1 {struct S s; int a;};         /* sizeof(struct S1) == 32  */
#pragma pack(2)
struct S2 {struct S s; int a;} s2;      /* sizeof(struct S2) == 32  */
                                        /* offsetof(S2, s1) == 0    */
                                        /* offsetof(S2, a) == 16    */
```

3. **#pragma pack**

   In this example, since the data types are by default packed along boundaries smaller than those specified by #pragma pack (8), they are still aligned along the smaller boundary (alignof(S2) = 4).

```
#pragma pack(2)
struct S {                      /* sizeof(struct S) == 48  */
   char a;                      /* offsetof(S, a) == 0     */
   int* b;                      /* offsetof(S, b) == 16    */
   char c;                      /* offsetof(S, c) == 32    */
   short d;                     /* offsetof(S, d) == 34    */
} S;                            /* alignof(S) == 16        */

struct S1 {                     /* sizeof(struct S1) == 10 */
   char a;                      /* offsetof(S1, a) == 0    */
   int b;                       /* offsetof(S1, b) == 2    */
   char c;                      /* offsetof(S1, c) == 6    */
   short d;                     /* offsetof(S1, d) == 8    */
} S1;                           /* alignof(S1) == 2        */

#pragma pack(8)
struct S2 {                     /* sizeof(struct S2) == 12 */
   char a;                      /* offsetof(S2, a) == 0    */
   int b;                       /* offsetof(S2, b) == 4    */
   char c;                      /* offsetof(S2, c) == 8    */
   short d;                     /* offsetof(S2, d) == 10   */
} S2;                           /* alignof(S2) == 4        */
```

4. **PACKSTRUCT Compiler Option**

   If the following is compiled with PACK STRUCTURE set to 2:

```
struct S1 {                     /* sizeof(struct S1) == 10 */
   char a;                      /* offsetof(S1, a) == 0    */
   int b;                       /* offsetof(S1, b) == 2    */
   char c;                      /* offsetof(S1, c) == 6    */
   short d;                     /* offsetof(S1, d) == 8    */
} S1;                           /* alignof(S1) == 2        */
```

5. **#pragma pack**

   If the following is compiled with PACK STRUCTURE set to 4:

```
#pragma pack(1)
struct A {              // this structure is packed along 1-byte boundaries
```

```
   char a1;
   int a2;
};

#pragma pack(2)
struct B {              // this class is packed along 2-byte boundaries
   int b1;
   float b2;
   float b3;
};

#pragma pack(pop)       // this brings pack back to 1-byte boundaries
struct C {
   int c1;
   char c2;
   short c3;
};

#pragma pack(pop)       // this brings pack back to the compile option,
struct D {              // 4-byte boundaries
   int d1;
   char d2;
};
```

6. **__align**

```
int __align(16) varA;   /* varA is aligned on a 16-byte boundary */
```

7. **_Packed**

```
struct A {                      /* sizeof(A) == 24      */
   int a;                       /* offsetof(A, a) == 0  */
   long long b;                 /* offsetof(A, b) == 8  */
   short c;                     /* offsetof(A, c) == 16 */
   char d;                      /* offsetof(A, d) == 18 */
};

_Packed struct B {              /* sizeof(B) == 15    */
   int a;                       /* offsetof(B, a) == 0  */
   long long b;                 /* offsetof(B, b) == 4  */
   short c;                     /* offsetof(B, c) == 12 */
   char d;                      /* offsetof(B, d) == 14 */
};
```

Layout of struct A, where * = padding:

```
|a|a|a|a|*|*|*|*|b|b|b|b|b|b|b|b|c|c|d|*|*|*|*|*|
```

Layout of struct B, where * = padding:

```
|a|a|a|a|b|b|b|b|b|b|b|b|c|c|d|
```

8. **__alignof**

```
struct A {
   char a;
   short b;
};

struct B {
   char a;
   long b;
} varb;

int var;
```

In the code sample above:

- __alignof(struct A) = 2
- __alignof(struct B) = 4
- __alignof(var) = 4

- __alignof(varb.a) = 1

```
__align(16) struct A {
   int a;
   int b;
};

#pragma pack(1)
struct B {
   long a;
   long b;
};

struct C {
   struct {
      short a;
      int b;
   } varb;
} var;
```

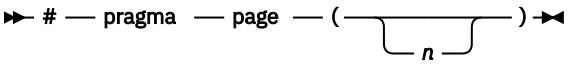
In the code sample above:

- __alignof(struct A) = 16
- __alignof(struct B) = 4
- __alignof(var) = 4
- __alignof(var.varb.a) = 4

page

C

page syntax



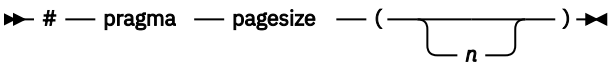
Description

Skips *n* pages of the generated source listing. If *n* is not specified, the next page is started.

pagesize

C

pagesize syntax



Description

Sets the number of lines per page to *n* for the generated source listing. The pagesize pragma may not affect the option listing page (sometimes called the Prolog).

pointer

C C++

pointer syntax

Description

Allows the use of IBM i pointer types:

- space pointer

- system pointer
- invocation pointer
- label pointer
- suspend pointer
- open pointer

A variable that is declared with a typedef that is named in the #pragma pointer directive has the pointer type associated with typedef_name in the directive. The <pointer.h> header file contains typedefs and #pragma directives for these pointer types. Including this header file in your source code allows you to use these typedefs directly for declaring pointer variables of these types.

Parameters

pointer_type
which can be one of:

SPCPTR
Space pointer

OPENPTR
Open pointer

SYSPTR
System pointer

INVPTR
Invocation pointer

LBLPTR
Label code pointer

SUSPENDPTR
Suspend pointer

Notes on Usage

The compiler issues a warning and ignores the #pragma pointer directive if any of the following errors occur:

- The pointer type that is named in the directive is not one of SPCPTR, SYSPTR, INVPTR, LBLPTR, SUSPENDPTR, or OPENPTR.
- The typedef named is not declared before the #pragma pointer directive.
- The identifier that is named as the first parameter of the directive is not a typedef.
- The typedef named is not a typedef of a void pointer.
- The typedef named is used in a declaration before the #pragma pointer directive.

The typedef named must be defined at file scope.

See the *ILE C/C++ Programmer's Guide* for more information about using IBM i pointers.

priority

C++

priority syntax

Description

The #pragma priority directive specifies the order in which static objects are to be initialized at runtime.

The value *n* is an integer literal in the range of `INT_MIN` to `INT_MAX`. The default value is 0. A negative value indicates a higher priority; a positive value indicates a lower priority.

The first 1024 priorities (INT_MIN to INT_MIN + 1023) are reserved for use by the compiler and its libraries. The #pragma priority can appear anywhere in the source file many times. However, the priority of each pragma must be greater than the previous pragma's priority. This is necessary to ensure that the runtime static initialization occurs in the declaration order.

Example

```
//File one called First.C

#pragma priority (1000)
class A { public: int a; A() {return;} } a;
#pragma priority (3000)
class C { public: int c; C() {return;} } c;
class B { public: int b; B() {return;} };
extern B b;
main()
{
    a.a=0;
    b.b=0;
    c.c=0;
}

//File two called Second.C
#pragma priority (2000)
class B { public: int b; B() {return;} } b;
```

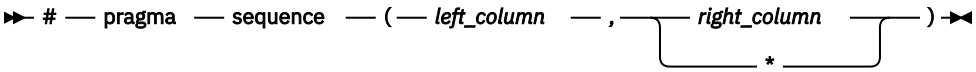
In this example, the execution sequence of the runtime static initialization is:

1. Static initialization with priority 1000 from file First.C
2. Static initialization with priority 2000 from file Second.C
3. Static initialization with priority 3000 from file First.C

sequence

C

sequence syntax



Description

Specifies the columns of the input record that are to contain sequence numbers. The column setting applies only to the source setting in which it is located and has no effect on any source members named on include directives in the member.

Parameters

left column
Must be greater than zero but less than 32 754. The *left column* should be less than the *right column*.

right column
Must be greater than zero but less than 32 754. The *right column* should be greater than or equal to the *left column*. An asterisk (*) that is specified as the *right column* value indicates that sequence numbers are contained between *left column* and the end of the input record.

Notes on Usage

The #pragma sequence directive takes effect on the line following the directive. It remains in effect until another #pragma sequence or #pragma nosequence directive is encountered or the end of the source member is reached.

The #pragma margins and #pragma sequence directives can be used together. If these two #pragma directives reserve the same columns, the #pragma sequence directive has priority, and the columns are reserved for sequence numbers.

For example, if the #pragma margins directive specifies margins of 1 and 20 and the #pragma sequence directive specifies columns 15 to 25 for sequence numbers, the margins in effect are 1 and 14, and the columns reserved for sequence numbers are 15 to 25.

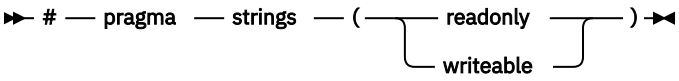
If the margins specified are not in the supported range or the margins contain non-numeric values, a warning message is issued during compilation and the directive is ignored.

See also pragmas "nosequence" on page 50 and "margins" on page 47.

strings

C C++

strings syntax



Description

Specifies that the compiler may place strings into read-only memory or must place strings into writeable memory. Strings are writeable by default. This pragma must appear before any C or C++ code in a file.

Note : This pragma will override the *STRDONLY compiler option.

weak

C++

weak syntax

Description

Identifies an identifier to the compiler as being a weak global symbol.

Parameters

identifier
Specifies the name of an identifier considered to be a weak global symbol.

identifier2
If *identifer2* is specified, then *identifier* is considered to be a weak global symbol whose value is the same as *identifier2*. For this pragma to have effect, *identifier2* must be defined in the same compilation unit.

This pragma can appear anywhere in a program, and identifies a specified identifier as being a weak global symbol. *Identifier* should not be defined, but it may be declared. If it is declared, and *identifier2* is specified, *identifier* must be of a type compatible to that of *identifier2*.

Example.

```
#pragma weak func1 = func2
```

Control Language Commands

Read this section for an overview of the Control Language (CL) commands that are used with the ILE C/C++ compiler. Syntax diagrams and parameter description tables are provided.

This table describes the CL commands that are used with the IBM i compiler.

Table 3. Control Language Commands

Action	Command	Description
Create C Module	CRTCMOD	Creates a module object (*MODULE) based on the source you provide.
Create C++ Module	CRTCPPMOD	
Create Bound C Program	CRTBNDC	Creates a program object (*PGM) based on the source you provide.
Create Bound C++ Program	CRTBNDCPP	

CL commands and their parameters can be entered in either uppercase or lowercase. In this reference, they are always shown in uppercase. For example:

```
CRTCPPMOD MODULE(ABC/HELLO) SRCSTMF('/home/usr/hello.C') OPTIMIZE(40)
```

ILE C/C++ language statements must be entered exactly as shown. For example, fopen, _Ropen, because the ILE C/C++ compiler is case sensitive.

Variables appear in lowercase italic letters, for example, *file-name*, *characters*, and *string*. They represent user-supplied names or values.

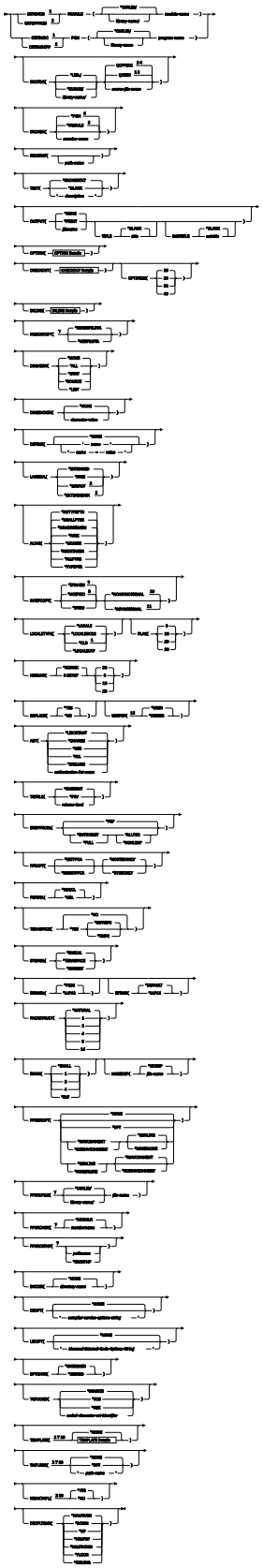
Language statements may contain punctuation marks, parentheses, arithmetic operators, or other such symbols. You must enter them exactly as shown in the syntax diagram.

You can also invoke the compiler and its options through the Qshell command line environment. For more information about Qshell command and option formats, see “Using the ixlc Command to Invoke the C/C++ Compiler” on page 103.

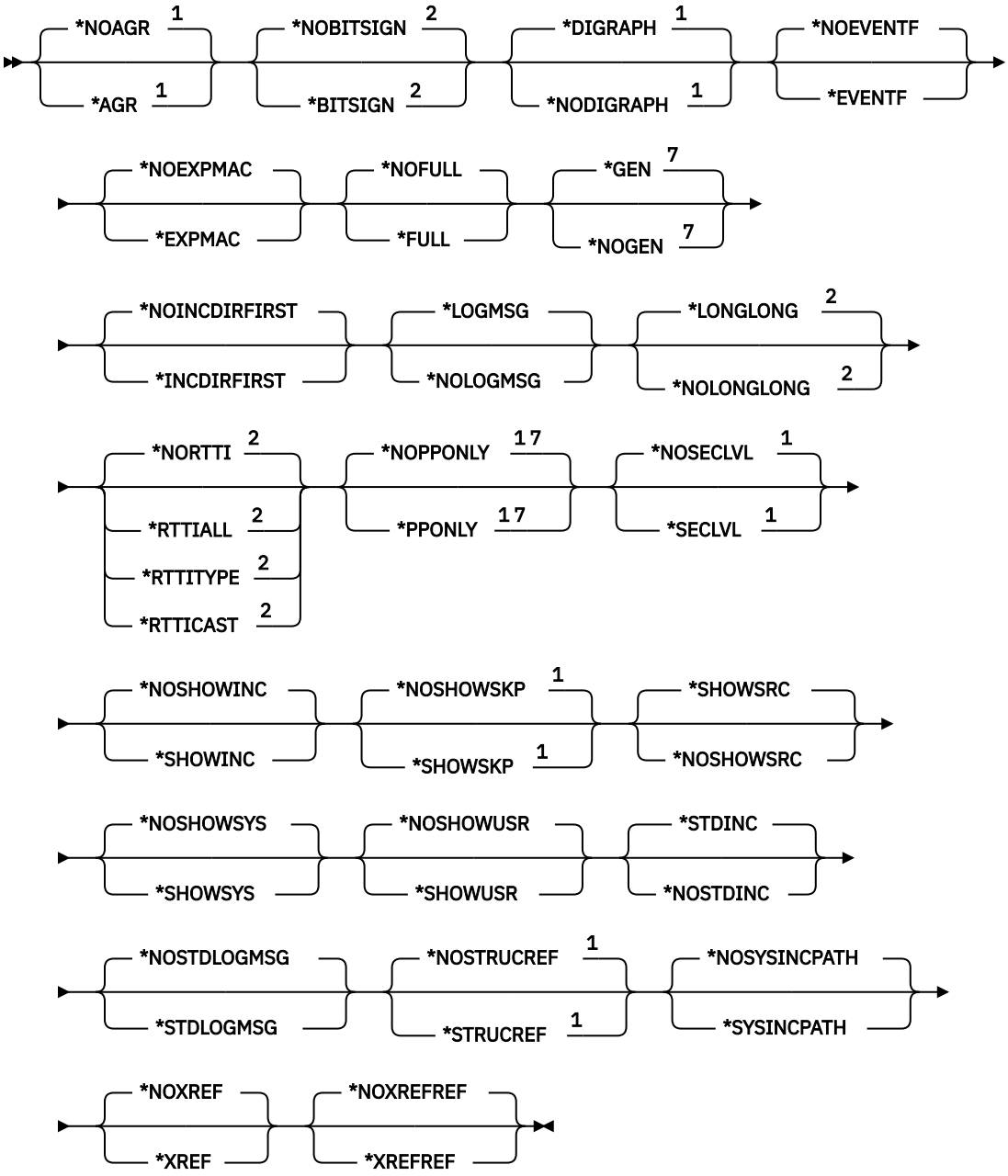
Control Language Command Syntax

The syntax diagrams in this section show all parameters and options of the CRTCMOD, CRTCPPMOD, CRTBNDC, and CRTBNDCPP commands, and the default values for each option. In most cases the keywords are identical for any of the commands. Differences are noted where they exist. For detailed descriptions of each option, see “Control Language Command Options” on page 65.

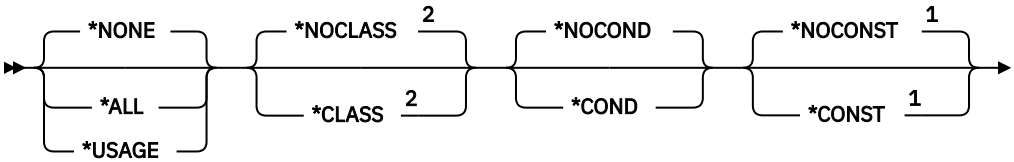
Syntax diagram

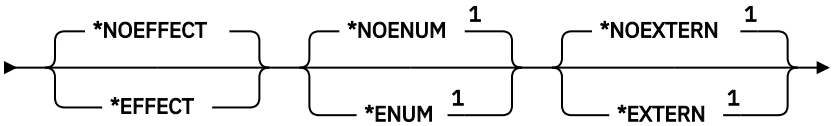


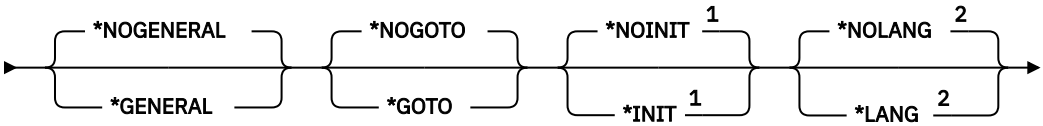
OPTION Details

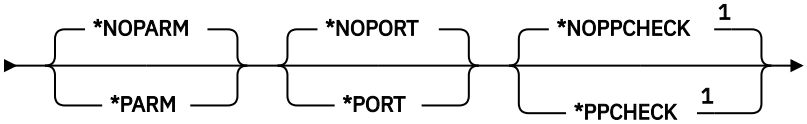


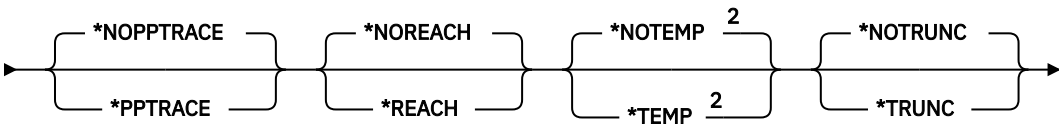
CHECKOUT Details

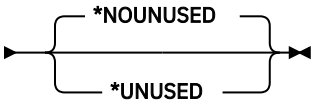




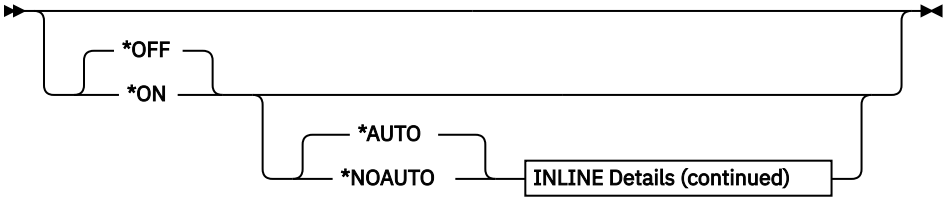




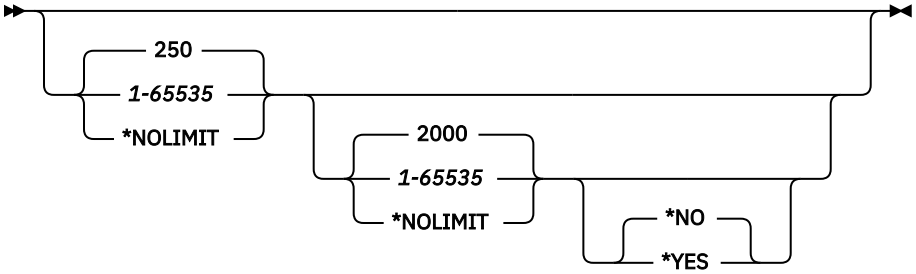




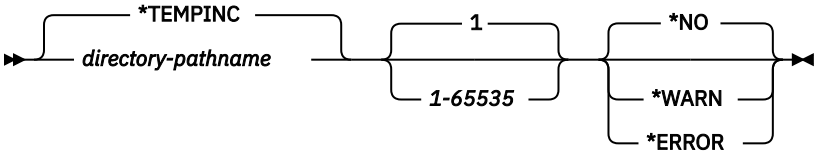
INLINE Details



INLINE Details (continued)



TEMPLATE Details



Notes:

[1] C compiler only

[2] C++ compiler only
[3] C compiler default setting
[4] C++ compiler default setting
[5] Create Module command only
[6] Create Bound Program command only
[7] Create module command only
[8] C compiler default setting
[9] C++ compiler default setting
[10] C compiler only
[11] C compiler only
[12] Create Bound Program command only
[13] Applicable only when using the Integrated File System (IFS)

Control Language Command Options

The following pages describe the keywords for the CRTCMOD, CRTCPPMOD, CRTBNDC, and CRTBNDCPP commands. In most cases the keywords are identical for any of the commands. Differences are noted where they exist.

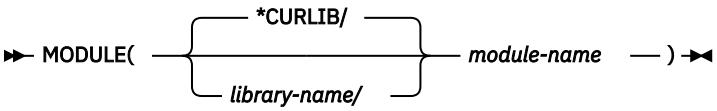
The term object is used throughout the descriptions and has one of two meanings:

- If you are using the CRTCMOD or CRTCPPMOD commands, object means module object.
- If you are using the CRTBNDC or CRTBNDCPP commands, object means program object.

MODULE

Valid only on the CRTCMOD and CRTCPPMOD commands. Specifies the module name and library for the compiled ILE C or C++ module object.

MODULE Syntax



***CURLIB**
This is the default library value. The object is stored in the current library. If a job does not have a current library, QGPL is used.

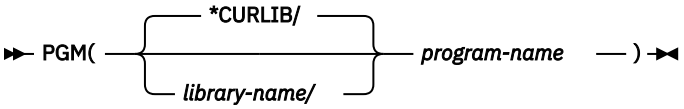
library-name
Enter the name of the library where the object is to be stored.

module-name
Enter a name for the module object.

PGM

Valid only on the CRTBNDC and CRTBNDCPP commands. Specifies the program name and library for the compiled ILE C or C++ program object.

PGM Syntax



***CURLIB**
This is the default library value. The object is stored in the current library. If a job does not have a current library, QGPL is used.

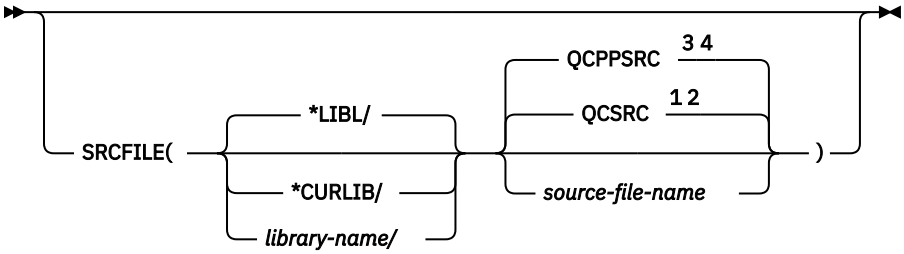
library-name
Enter the name of the library where the object is to be stored.

program-name
Enter a name for the program object.

SRCFILE

Specifies the source physical file name and library of the file that contains the ILE C or C++ source code that you want to compile.

SRCFILE Syntax



Notes:

1 C Compiler only
2 C Compiler default setting
3 C++ Compiler only
4 C++ Compiler default setting

***LIBL**
This is the default library value. The library list is searched to find the library where the source file is located.

***CURLIB**
The current library is searched for the source file. If a job does not have a current library, QGPL is used.

library-name
Enter the name of the library that contains the source file.

QCSRC C
The default name for the source physical file that contains the member with the ILE C source code that you want to compile.

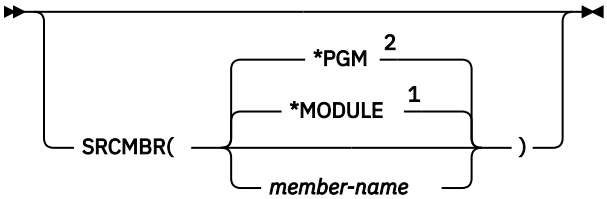
QCPPSRC C++
The default name for the source physical file that contains the member with the ILE C++ source code that you want to compile.

source-file-name
Enter the name of the file that contains the member with the ILE C or C++ source code.

SRCMBR

Specifies the name of the member that contains the ILE C or C++ source code.

SRCMBR Syntax



Notes:

[1] Create Module command only

[2] Create Bound Program command only

***MODULE**

Valid only with the CRTCMOD or CRTCPPMOD commands. The module name that is supplied on the MODULE parameter is used as the source member name. This is the default when a member name is not specified.

***PGM**

Valid only with the CRTBNDC or CRTBNDCPP commands. The program name that is supplied on the PGM parameter is used as the source member name. This is the default when a member name is not specified.

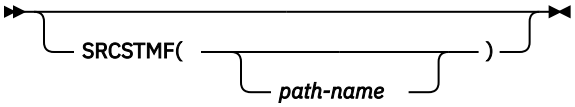
member-name

Enter the name of the member that contains the ILE C or C++ source code.

SRCSTMF

Specifies the path name of the stream file containing the ILE C or C++ source code that you want to compile.

SRCSTMF Syntax



The path name can be either absolutely or relatively qualified. An absolute path name starts with '/'; a relative path name starts with a character other than '/'. If absolutely qualified, then the path name is complete. If relatively qualified, the path name is completed by pre-pending the job's current working directory to the path name.

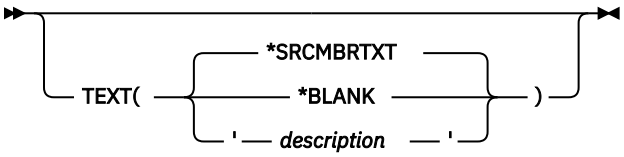
Note :

1. The SRCMBR and SRCFILE parameters cannot be specified with the SRCSTMF parameter.
2. If SRCSTMF is specified, then the following compiler options are ignored:
   - TEXT(*SRCMBRTXT)
   - OPTION(*STDINC)
   - OPTION(*SYSINCPATH)
3. The SRCSTMF parameter is not supported in a mixed-byte environment.

TEXT

Allows you to enter text that describes the object and its function.

TEXT Syntax



***SRCMBRTXT**

Default setting. The text description that is associated with the source file member is used for the compiled object. If the source file is an inline file or a device file, this field is blank.

***BLANK**

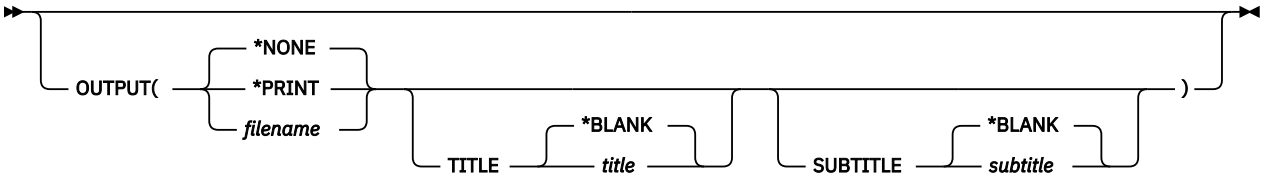
Specifies that no text appears.

description
Enter descriptive text no longer than 50 characters, and enclose it in single quotation marks. The quotation marks are not part of the 50-character string. Quotation marks are supplied when the CRTCMOD or CRTCPPMOD prompt screens are used.

OUTPUT

Specifies if the compiler listing is required or not.

OUTPUT Syntax



***NONE**
Does not generate the compiler listing. When a listing is not required, use this default to improve compile time performance. When *NONE is specified, the following listing-related options are ignored if they are specified on the OPTION keyword: *AGR, *EXPMAC, *FULL, *SECLVL, *SHOWINC, *SHOWSKP, *SHWSRC, *SHOWSYS, *SHOWUSR, *SHWSRC, *STRUCREF, *XREF, or *XREFREF.

***PRINT**
Generate the compiler listing as a spool file.

The spool file name in WRKSPLF will have the same name as the object (program or module) being created.

filename
The compiler listing is saved in the file name specified by this string.

The listing name must be in Integrated File System (IFS) format, for example **/home/mylib/listing/hello.lst**. A data management file *listing* in library *mylib* should be specified as **/QSYS.LIB/***mylib*.lib/*listing*.file/hello.mbr. If the string does not begin with a "/", it will be considered a subdirectory of the current directory or library. If the file does not exist, the file will be created.

Data authority *WX is required to create an IFS listing. Data authority *WX, object authority *OBJEXIST and *OBJALTER are required to create a data management file listing via IFS.

TITLE
Specifies the title for the compiler listing. Possible TITLE values are:

***BLANK**
No title is generated.

title
Specify a title string (maximum 98 characters) for the listing.

SUBTITLE
Specifies the subtitle for the compiler listing. Possible SUBTITLE values are:

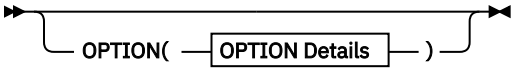
***BLANK**
No title is generated.

subtitle
Specify a subtitle string (maximum 98 characters) for the listing file.

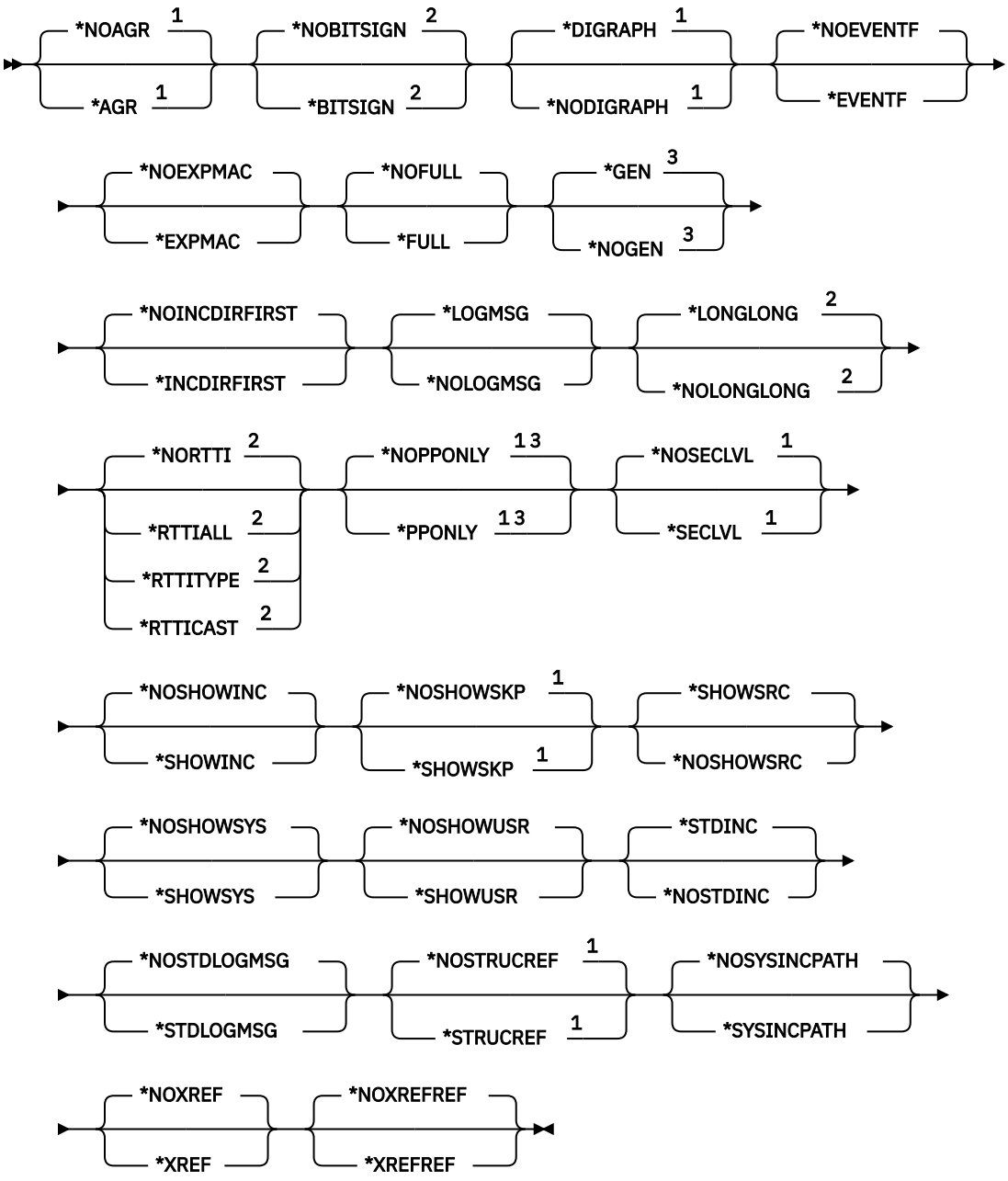
OPTION

Specifies the options to use when the ILE C or C++ source code is compiled. You can specify them in any order, separated by a blank space. Unless noted otherwise in the option descriptions, when an option is specified more than once, or when two options conflict, the last one that is specified is used.

OPTION Syntax



OPTION Details



Notes:

[1] C compiler only
[2] C++ compiler only
[3] Create Module command only

The possible options are:

***NOAGR** C

Accepted but ignored by the C++ compiler. Default setting. Does not generate an aggregate structure map in the compiler listing.

***AGR** C

Accepted but ignored by the C++ compiler. Generates an aggregate structure map in the compiler listing. This map provides the layout of all structures in the source program, and shows whether variables are padded or not. OUTPUT(*PRINT) must be specified.

The *AGR option overrides the *STRUCREF option.

***NOBITSIGN** C++

Default setting. Bitfields are unsigned.

***BITSIGN** C++

Bitfields are signed.

***NODIGRAPH** C

Default setting. Digraph character sequences are not recognized by the compiler. Syntax errors might result if digraphs are encountered with this setting in effect.

***DIGRAPH** C

Digraph character sequences can be used to represent characters not found on some keyboards. Digraph character sequences appearing in character or string literals are not replaced during preprocessing.

***NOEVENTF**

Default setting. Does not create an event file for use by Cooperative Development Environment/400 (CODE/400).

***EVENTF**

Creates an event file for use by Cooperative Development Environment/400 (CODE/400). The event file is created as a member in file EVFEVENT in the library where the created module or program object is to be stored. If the file EVFEVENT does not exist, it is automatically created. The event file member name is the same as the name of the object being created. An Event File is normally created when you create a module or program from within CODE/400. CODE/400 uses this file to provide error feedback integrated with the CODE/400 editor.

***NOEXPMAC**

Default setting. Does not expand the macros in the source section of the listing or in the debug listing view.

***EXPMAC**

Expands all macros in the source section of a listing view. If this suboption is specified together with DBGVIEW(*ALL) or DBGVIEW(*LIST), the compiler issues an error message and stops compilation.

***NOFULL**

Default setting. Does not show all compiler-output information in the listing or in the debug listing view.

***FULL**

Shows all compiler-output information in the listing or in the debug listing view. This setting turns on all listing-related options. If *FULL is specified, you can turn off an individual listing option by specifying the *NO setting for that option after the *FULL option. If this suboption is specified together with DBGVIEW(*ALL) or DBGVIEW(*LIST), the compiler issues an error message and stops compilation.

***GEN**

Valid only with the CRTCMOD and CRTCPPMOD commands. Default setting. All phases of the compilation process are carried out.

C Specifying OPTION(*PPONLY) overrides the PPGENOPT(*NONE) and OPTION(*GEN) option settings. Instead, the following settings are implied:

- PPGENOPT(*DFT) PPSRCFILE(QTEMP/QACZEXPAND) PPSRCMBR(*MODULE) for a data management source file, or,
- PPGENOPT(*DFT) PPSRCSTMF(*SRCSTMF) for an IFS source file.

***NOGEN**
Valid only with the CRTCMOD and CRTCPPMOD commands. Compilation stops after syntax checking. No object is created.

***NOINCDIRFIRST**
Default setting. The compiler searches for user include files in the root source directory first, and then in the directories specified by the INCDIR option.

***INCDIRFIRST**
The compiler searches for user include files as follows:

1. If you specify a directory in the INCDIR parameter, the compiler searches for *file_name* in that directory.
2. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line.
3. Searches the directory where your current root source file resides.
4. If the INCLUDE environment variable is defined, the compiler searches the directories in the order they appear in the INCLUDE path.
5. If the *NOSTDINC compiler option is not chosen, search the default include directory /QIBM/include.

***LOGMSG**
Default setting. Compilation messages are put into the job log.

When you specify this option and the FLAG parameter, messages with the severity specified on the FLAG parameter (and higher) are placed in the job log.

When you specify this option and a maximum number of messages on the MSGLMT parameter, compilation stops when the number of messages, at the specified severity, have been placed in the job log.

***NOLOGMSG**
Does not put the compilation messages into the job log.

***LONGLONG** C++
Default setting. The compiler recognizes and uses the longlong data type.

***NOLONGLONG** C++
The compiler does not recognize the longlong data type.

***NORTTI** C++
Default setting. The compiler does not generate information needed for RunTime Type Information (RTTI) typeid and dynamic_cast operators.

***RTTIALL** C++
The compiler generates the information needed for the RTTI typeid and dynamic_cast operators.

***RTTITYPE** C++
The compiler generates the information needed for the RTTI typeid operator, but the information for the dynamic_cast operator is not generated. This option is not supported if the RTBND(*LLP64) compile option is specified.

***RTTICAST** C++
The compiler generates the information needed for the RTTI dynamic_cast operator, but the information for the typeid operator is not generated. This option is not supported if the RTBND(*LLP64) compile option is specified.

***NOPPONLY** C
Valid only with the CRTCMOD command. Default setting. The compiler runs the entire compile sequence when *GEN is left as the default for OPTION.

Specifying PPGENOPT with any setting other than *NONE overrides the OPTION(*NOPPONLY) and OPTION(*GEN) option settings.

Note : The PPGENOPT compiler option replaces OPTION(*NOPPONLY). Support for OPTION(*NOPPONLY) may be removed in future releases.

***PPONLY** C

Valid only with the CRTCMOD command. The preprocessor is run and the output is saved in the source file QACZEXPAND in library QTEMP. The member-name is the same as the name specified on the MODULE parameter. The rest of the compilation sequence is not run. When the job is submitted in batch mode, the output is deleted once the job is complete.

If you specify SRCSTMF, then the compiler saves the output in a stream file in your current directory. The file name is the same as the file on SRCSTMF with a ".i" extension.

Specifying OPTION(*PPONLY) overrides the PPGENOPT(*NONE) and OPTION(*GEN) option settings. Instead, the following settings are implied:

- PPGENOPT(*DFT) PPSRCFILE(QTEMP/QACZEXPAND) PPSRCMBR(*MODULE) for a data management source file, or,
- PPGENOPT(*DFT) PPSRCSTMF(*SRCSTMF) for an IFS source file.

Note : The PPGENOPT compiler option replaces OPTION(*PPONLY). Support for OPTION(*PPONLY) might be removed in future releases.

***NOSECLVL** C

Default setting. Does not generate the second-level message text in the listing.

***SECLVL** C

Generates the second-level message text in the listing. OUTPUT(*PRINT) must be specified.

***NOSHOWINC**

Default setting. Does not expand the user include files or the system include files in the source listing or in the debug listing view.

***SHOWINC**

Expands both the user-include files and the system-include files in the source section of the listing or in the debug listing view. OUTPUT(*PRINT) or DBGVIEW(*ALL, *SOURCE, or *LIST) must be specified.

This setting turns on the *SHOWUSR and *SHOWSYS settings, but those settings can be overridden by specifying *NOSHOWUSR or *NOSHOWSYS or both after *SHOWINC.

***NOSHOWSKP** C

Default setting. Does not include the statements that the preprocessor has ignored in the source section of the listing or in the debug listing view. The preprocessor ignores statements as a result of a preprocessor directive evaluating to false (zero).

***SHOWSKP** C

Includes all statements in the source listing or in the debug listing view, regardless of whether the preprocessor has skipped them. OUTPUT(*PRINT) or DBGVIEW(*ALL or *LIST) must be specified.

***SHOWSRC**

Default setting. Shows the source statements in the source listing or in the debug listing view. OUTPUT(*PRINT) or DBGVIEW(*ALL, *SOURCE, or *LIST) must be specified.

***NOSHOWSRC**

Does not show the source statements in the source listing or in the debug listing view. The *EXPMAC,*SHOWINC,*SHOWUSR,* *SHOWSKP listing options can override this setting if specified after the*NOSHOWSRC option.

***NOSHOWSYS**

Default setting. Does not expand the system include files on the #include directive in the source listing or in the debug listing view.

***SHOWSYS**

Expands the system include files on the #include directive in the source listing or in the debug listing view. An OUTPUT option, or DBGVIEW parameter value of *ALL, *SOURCE or *LIST must be specified. System include files on the #include directive are enclosed in angle brackets (< >).

***NOSHOWUSR**
Default setting. Does not expand the user include files on the #include directive in the source listing or in the debug listing view.

***SHOWUSR**
Expands the user include files on the #include directive in the source listing or in the debug listing view. OUTPUT(*PRINT) or DBGVIEW(*ALL, *SOURCE, or *LIST) must be specified. User-include files on the #include directive are enclosed in double quotation marks (" "). Use this option to print the typedef that is generated when you use #pragma mapinc in your ILE C or C++ program to process externally described files.

***STDINC**
Default setting. The compiler includes the default include path (/QIBM/include for IFS source stream files; QSYSINC for data management source file members) at the end of the search order.

***NOSTDINC**
The compiler removes the default include path (/QIBM/include for IFS source stream files; QSYSINC for data management source file members) from the search order.

***NOSTDLOGMSG**
Default setting. The compiler does not produce stdout compiler messages.

***STDLOGMSG**
The compiler produces stdout compiler messages when working in the Qshell environment. This option has no effect when compiling with TGTRLS(*PRV).

***NOSTRUCREF** 
Default setting. Does not generate an aggregate structure map of all referenced struct or union variables in the compiler listing.

***STRUCREF** 
Generates an aggregate structure map of all referenced struct or union variables in the compiler listing. This map provides the layout of all referenced structures in the source program, and shows whether variables are padded or not.

***NOSYSINCPATH**
Default setting. The search path for user includes is not affected.

***SYSINCPATH**
Changes the search path of user includes to the system include search path. In function this option is equivalent to changing the double-quotes in the user #include directive (#include "file_name") to angle brackets (#include <file_name>).

***NOXREF**
Does not generate the cross-reference table in the listing. *NOXREF is the default.

***XREF**
Generates the cross-reference table that contains a list of the identifiers in the source code together with the line number in which they appear. An OUTPUT option must be specified.

The *XREF option overrides the *XREFREF option.

***NOXREFREF**
Default setting. Does not generate the cross-reference table in the listing.

***XREFREF**
Generates the cross-reference table, including only referenced identifers and variables in the source code, together with the line number in which they appear. An OUTPUT option must be specified.

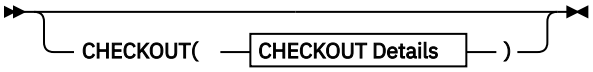
The *XREF option overrides the *XREFREF option.

CHECKOUT

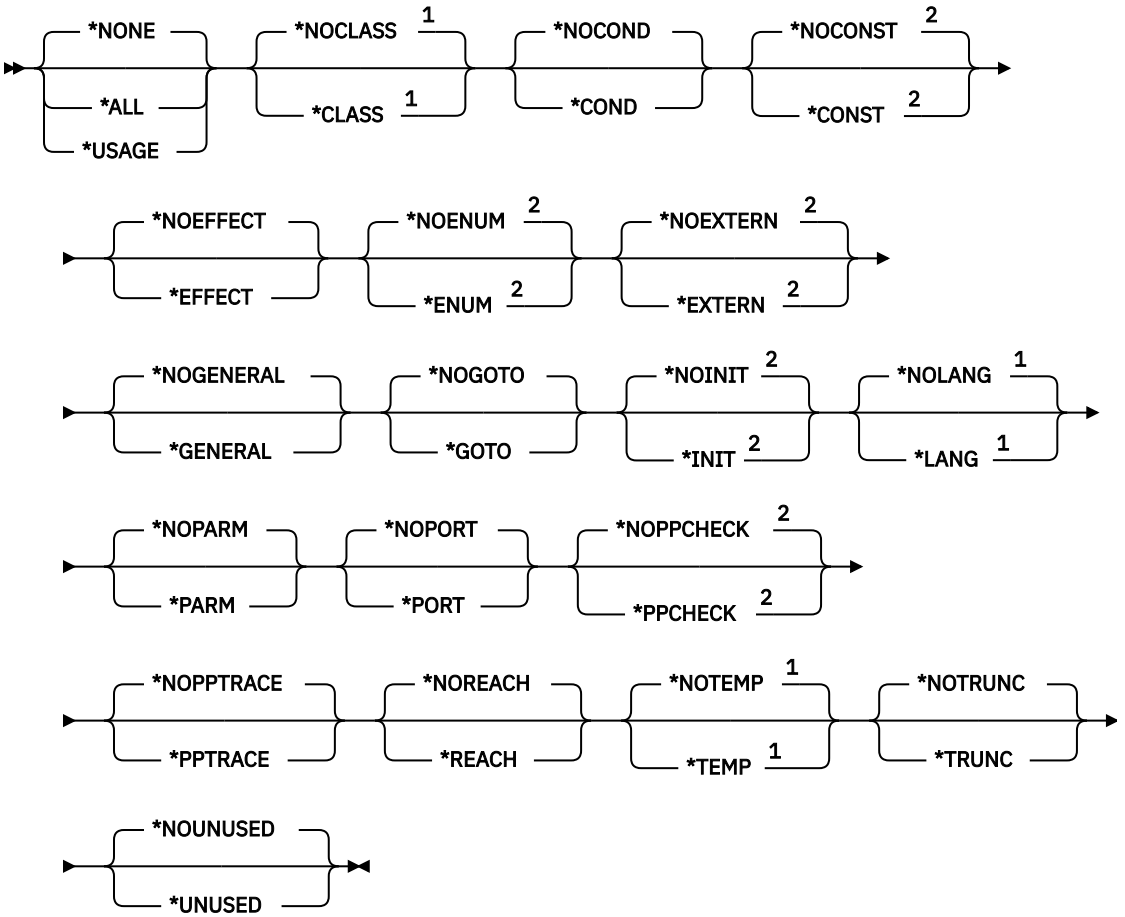
Specifies options you may select to generate informational messages that indicate possible programming errors. When you specify an option more than once, or when two options conflict, the last one that is specified is used.

Note : CHECKOUT may produce many messages. To prevent these messages from going to the job log specify OPTION(*NOLOGMSG) and the source listing option OUTPUT(*PRINT).

CHECKOUT Syntax



CHECKOUT Details



Notes:

[1] C++ compiler only
[2] C compiler only

The possible options are:

***NONE**
Default setting. Disables all of the options for CHECKOUT.

***ALL**
Enables all of the options for CHECKOUT.

***USAGE**

- C Equivalent to specifying *ENUM, *EXTERN, *INIT, *PARM, *PORT, *GENERAL, and *TRUNC. All other CHECKOUT options are disabled.
- C++ Equivalent to specifying *COND. All other CHECKOUT options are disabled.

***NOCLASS** C++
Default setting. Does not display info about class use.

***CLASS** C++
Display info about class use.

***NOCOND**
Default setting. Does not warn about possible redundancies or problems in conditional expressions.

***COND**
Warn about possible redundancies or problems in conditional expressions.

***NOCONST** C
Default setting. Does not warn about operations involving constants.

***CONST** C
Warn about operations involving constants.

***NOEFFECT**
Default setting. Does not warn about statements with no effect.

***EFFECT**
Warn about statements with no effect.

***NOENUM** C
Default setting. Does not list the usage of enumerations.

***ENUM** C
Lists the usage of enumerations.

***NOEXTERN** C
Default setting. Does not list the unused variables that have external declarations.

***EXTERN** C
Lists the unused variables that have external declarations.

***NOGENERAL**
Default setting. Does not list the general CHECKOUT messages.

***GENERAL**
Lists the general CHECKOUT messages.

***NOGOTO**
Default setting. Does not list the occurrence and usage of goto statements.

***GOTO**
Lists the occurrence and usage of goto statements.

***NOINIT** C
Default setting. Does not list the automatic variables that are not explicitly initialized.

***INIT** C
Lists the automatic variables that are not explicitly initialized.

***NOLANG** C++
Default setting. Does not display information about the effects of the language level.

***LANG** C++
Display information about the effects of the language level.

***NOPARM**
Default setting. Does not list the function parameters that are not used.

***PARM**
Lists the function parameters that are not used.

***NOPORT**
Default setting. Does not list the non-portable usage of the C or C++ language.

***PORT**
Lists the non-portable usage of the C or C++ language.

***NOPPCHECK** C
Default setting. Does not list the preprocessor directives.

***PPCHECK** C
Lists all preprocessor directives.

***NOPPTRACE**
Default setting. Does not list the tracing of include files by the preprocessor.

***PPTRACE**
Lists the tracing of include files by the preprocessor.

***NOREACH**
Default setting. Does not warn about unreachable statements.

***REACH**
Warn about unreachable statements.

***NOTEMP** C++
Default setting. Does not display information about temporary variables.

***TEMP** C++
Display information about temporary variables.

***NOTRUNC**
Default setting. Does not warn about the possible truncation or loss of data.

***TRUNC**
Warn about the possible truncation or loss of data.

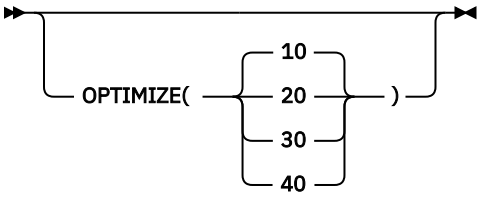
***NOUNUSED**
Default setting. Does not check for unused auto or static variables.

***UNUSED**
Check for unused auto or static variables.

OPTIMIZE

Specifies the level of the object's optimization.

OPTIMIZE Syntax



10
Default setting. Generated code is not optimized. This level has the shortest compile time.

20
Some optimization is performed on the code.

30
Full optimization is performed on the generated code.

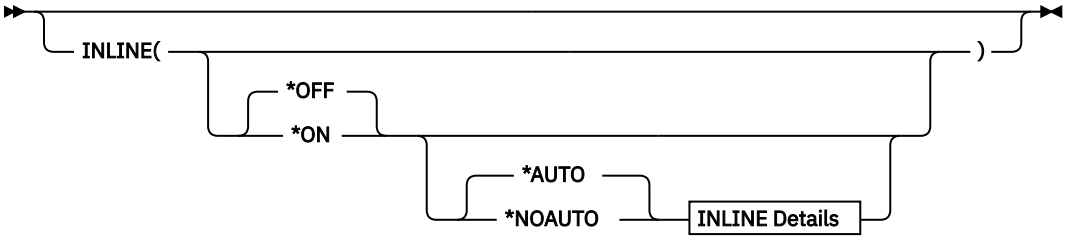
40
All optimizations done at level 30 are performed on the generated code. In addition, code is eliminated from procedure prologues and epilogues that enables instruction trace and call trace system functions. Eliminating this code enables the creation of leaf procedures. A leaf procedure contains no calls to other procedures. Procedure call performance to a leaf procedure is significantly faster than to a normal procedure.

INLINE

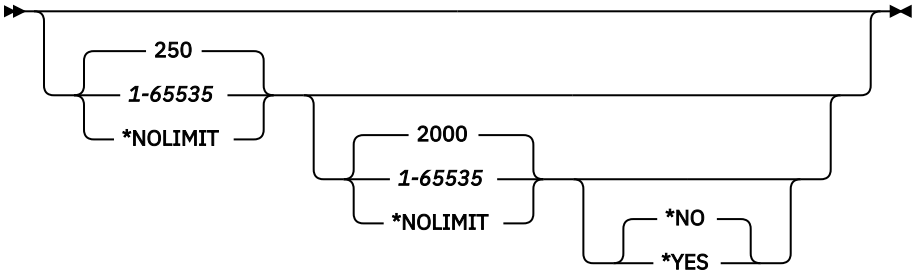
Allows the compiler to consider replacing a function call with the called function's instructions. Inlining a function eliminates the overhead of a call and can result in better optimization. Small functions that are called many times are good candidates for inlining.

Note : When specifying an INLINE option, all preceding INLINE options must also be specified, including their defaults.

INLINE Syntax



INLINE Details (continued)



The possible INLINE options are:

Inliner

Specifies whether inlining is to be used.

***OFF**

Default setting. Specifies that inlining will not be performed on the compilation unit.

***ON**

Specifies that inlining will be performed on the compilation unit. If a debug listing view is specified, the inliner is turned off.

Mode

Specifies whether the inliner should attempt to automatically inline functions depending on their Threshold and Limit.

***AUTO**

Specifies that the inliner should determine if a function can be inlined based on the specified Threshold and Limit. The #pragma noinline directive overrides *AUTO. This is the default.

***NOAUTO**

Specifies that only the functions that have been marked for inlining should be considered candidates for inlining. Functions marked for inlining include C functions for which the #pragma inline directive was specified, C++ functions declared with the inline keyword, and C++ functions marked for inlining by language rules.

Threshold

Specifies the maximum size of a function that can be a candidate for automatic inlining. The size is measured in Abstract Code Units. Abstract Code Units are proportional in size to the executable code in the function; C and C++ code is translated into Abstract Code Units by the compiler.

250

Specifies a threshold of 250. This is the default.

1-65535

Specifies a threshold from 1 to 65535.

***NOLIMIT**

Defines the threshold as the maximum size of the program.

Limit
Specifies the maximum relative size a function can grow before auto-inlining stops.

2000
Specifies a limit of 2000. This is the default.

1-65535
Specifies a limit from 1 to 65535.

***NOLIMIT**
Limit is defined as the maximum size of the program. System limits may be encountered.

Report
Specifies whether to produce an inliner report with the compiler listing.

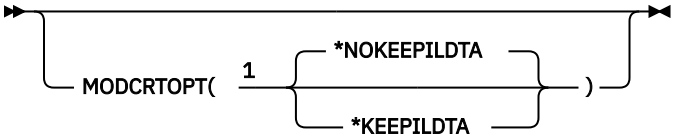
***NO**
The inliner report is not produced. This is the default.

***YES**
The inliner report is produced. OUTPUT(*PRINT) must be specified to produce the inliner report.

MODCRTOPT

Valid only with the CRTCMOD and CRTCPPMOD commands. Specifies the options to use when the *MODULE object is created. You can specify these options in any order, separated by spaces. When an option is specified more than once, or when two options conflict, the last one specified is used.

MODCRTOPT Syntax



Notes:

[1] Create Module command only

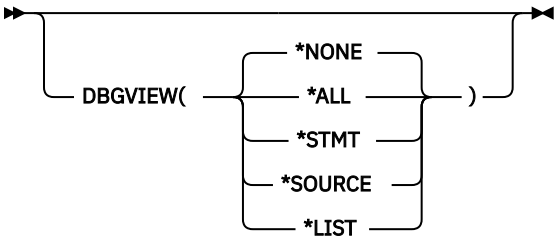
***NOKEEPILDTA**
Default setting. Intermediate language data is not stored with the *MODULE object.

***KEEPILDTA**
Intermediate language data is stored with the *MODULE object.

DBGVIEW

Specifies which level of debugging is available for the created program object. It also specifies which source views are available for source-level debugging. Requesting a debug listing view will turn inlining off.

DBGVIEW Syntax



The possible options are:

***NONE**
Default setting. Disables all of the debug options for debugging the compiled object.

***ALL**
Enables all of the debug options for debugging the compiled object and produces a source view, as well as a listing view. If this suboption is specified together with OPTION(*FULL) or OPTION(*EXPMAC), the compiler issues an error message and stops compilation.

***STMT**
Allows the compiled object to be debugged using program statement numbers and symbolic identifiers.

Note : To debug an object using the *STMT option you need a spool file listing.

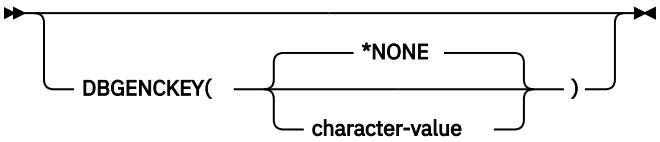
***SOURCE**
Generates the source view for debugging the compiled object. The OPTION(*NOSHOWINC, *SHOWINC, *SHOWSYS, *SHOWUSR) determines the content of the source view that is created.

Note : The root source should not be modified, renamed or moved after the module has been created. It must be in the same library/file/member, in order to use this view for debugging.

***LIST**
Generates the listing view for debugging the compiled object. The listing options (*EXPMAC, *NOEXPMAC, *SHOWINC, *SHOWUSR, *SHOWSYS, *NOSHOWINC, *SHOWSKP, *NOSHOWSKP) specified on the OPTION keyword determine the content of the listing view created, as well as the spool file listing. If this suboption is specified together with OPTION(*FULL) or OPTION(*EXPMAC), the compiler issues an error message and stops compilation.

DBGENCKEY

Specifies the encryption key to be used to encrypt program source that is embedded in debug views.
DBGENCKEY Syntax



The possible options are:

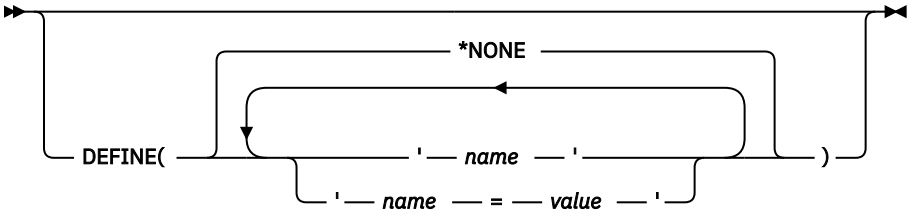
***NONE**
Default setting. No encryption key is specified.

character-value
Specify the key to be used to encrypt program source that is embedded in debug views stored in the module object. The length of the key can be between 1 byte and 16 bytes. A key of length 1 byte to 15 bytes is padded to 16 bytes with blanks for the encryption. Specifying a key of length zero is the same as specifying *NONE.

If the key contains any characters which are not invariant over all code pages, it is up to the user to ensure that the target system uses the same code page as the source system, otherwise the key might not match, and decryption might fail. If the encryption key must be entered on systems with differing code pages, it is recommended that the key is made of characters which are invariant for all EBCDIC code pages.

DEFINE

Specifies preprocessor macros that take effect before the file is processed by the compiler. The format DEFINE(macro) is equivalent to specifying DEFINE('macro=1').
DEFINE Syntax



***NONE**
Default setting. No macro is defined.

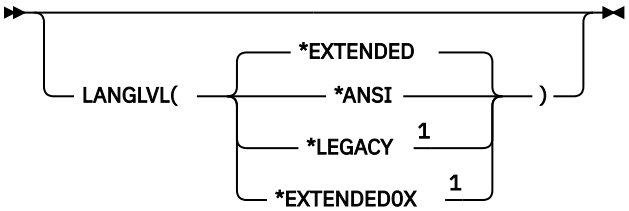
name* or *name=value
A maximum of 32 macros may be defined, and the maximum length of a macro is 80 characters. Enclose each macro in single quotation marks. The quotation marks are not part of the 80 character string and are not required when the CRTCMOD or CRTCPPMOD prompt screens are used. Single quotation marks are required for case-sensitive macros. Separate macros with blank spaces. If *value* is not specified, the compiler assigns a value of 1 to the macro.

Note : Macros, that are defined in the command, override any macro definition of the same name in the source. A warning message is generated by the compiler. Function-like macros such as #define max(a,b) ((a)>;(b):(a)?(b)) cannot be defined on the command.

LANGLVL

Specifies which group of language features are included when the source is compiled. When no LANGLVL is specified, the language level defaults to *EXTENDED.

LANGLVL Syntax



Notes:

[1] C++ compiler only

***EXTENDED**
Default setting. Defines the preprocessor variable __EXTENDED__ and undefines other language-level variables. ISO standard C and C++, and the IBM language extensions and system-specific features are available. This parameter should be used when all the functions of ILE C or C++ are to be available.

***ANSI**
Defines the preprocessor variables __ANSI__ and __STDC__ for C and C++ compilations, __cplusplus98__interface__ for C++ compilations only, and undefines other language-level variables. Only ISO standard C and C++ is available.

***LEGACY** C++
Undefines other language-level variables. Allow constructs compatible with older levels of the C++ language.

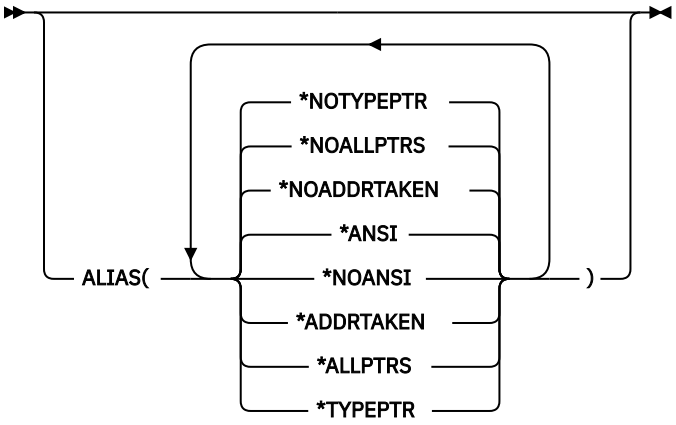
***EXTENDED0X** C++
Defines the same preprocessor variables as *EXTENDED does, and also defines an individual preprocessor variable for each C++11 language feature supported in this release. This option causes the compiler to use all the capabilities of ILE C++ and currently supported C++11 features that are implemented in this version of ILE C++ compiler. See “Extensions for C++11 compatibility” in *ILE C/C++ Language Reference*.

See also “ILE C/C++ Predefined Macros” on page 12

ALIAS

Indicates whether a program contains certain categories of aliasing or whether a program does not conform to C/C++ standard aliasing rules. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage location.

ALIAS Syntax



<u>*ANSI</u>|*NOANSI

When *ANSI is in effect, type-based aliasing is used during optimization, which restricts the lvalues that can be safely used to access a data object. The optimizer assumes that pointers can only point to an object of the same type.

When *NOANSI is in effect, the optimizer makes worst case aliasing assumptions. It assumes that a pointer of a given type can point to an external object or any object whose address is already taken, regardless of type.

***ADDRTAKEN|<u>*NOADDRTAKEN</u>**

When *ADDRTAKEN is in effect, variables are disjoint from pointers unless their address is taken. Any class of variable for which an address has not been recorded in the compilation unit will be considered disjoint from indirect access through pointers.

When *NOADDRTAKEN is specified, the compiler generates aliasing based on the aliasing rules that are in effect.

***ALLPTRS|<u>*NOALLPTRS</u>**

When *ALLPTRS is in effect, pointers are never aliased (this also implies *TYPEPTR). Specifying *ALLPTRS is an assertion to the compiler that no two pointers point to the same storage location. The suboption *ALLPTRS is only valid if *ANSI is also specified.

***TYPEPTR|<u>*NOTYPEPTR</u>**

When *TYPEPTR is in effect, pointers to different types are never aliased. Specifying *TYPEPTR is an assertion to the compiler that no two pointers of different types point to the same storage location. The suboption *TYPEPTR is only valid if *ANSI is also specified.

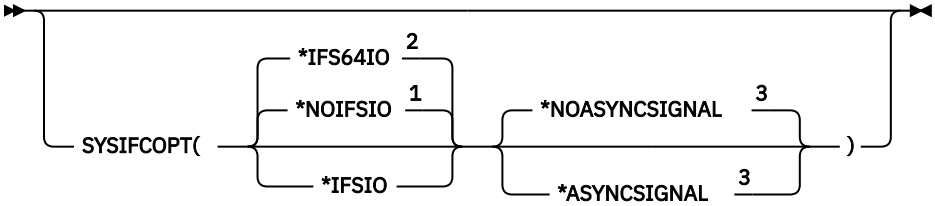
Note :

1. If conflicting ALIAS settings are specified, the last setting specified is used. For example, if ALIAS(*TYPEPTR *NOTYPEPTR) is specified, *NOTYPEPTR is used.
2. ALIAS makes assertions to the compiler about the code that is being compiled. If the assertions about the code are false, then the code generated by the compiler may result in unpredictable behaviour when the application is run.
3. The following are not subject to type-based aliasing.
   - Signed and unsigned types. For example, a pointer to a signed int can point to an unsigned int.
   - Character pointer types can point to any type
   - Types qualified as volatile or const. For example, a pointer to a const int can point to an int.

SYSIFCOPT

Specifies which integrated file system options will be used for C or C++ stream I/O operations in the module that is created.

SYSIFCOPT Syntax



Notes:

[1] C compiler default setting
[2] C++ compiler default setting
[3] C compiler only

***IFS64IO**
Default setting for the C++ compiler. The object that is created will use 64–bit Integrated File System APIs that support C and C++ stream I/O operations on files greater than two gigabytes in size. Using this option is equivalent to specifying SYSIFCOPT(*IFSIO *IFS64IO).

***NOIFSIO**
Default setting for the C compiler. The object that is created will use the IBM i Data Management file system for C and C++ stream I/O operations.

***IFSIO**
The object that is created will use the Integrated File System APIs for C and C++ stream I/O operations on files up to two gigabytes in size.

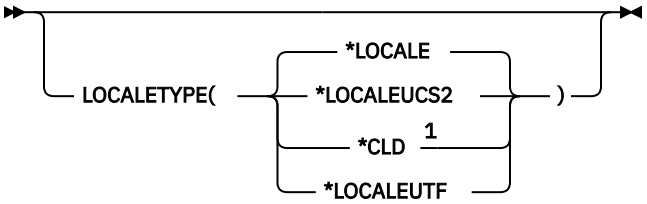
***NOASYNCSIGNAL** C
Default setting. Does not enable runtime mapping of synchronous signalling functions to asynchronous signalling functions.

***ASYNCSIGNAL** C
Enable runtime mapping of synchronous signalling functions to asynchronous signalling functions. Specifying this option causes C runtime environment to map the synchronous `signal()` function to the asynchronous `sigaction()` function, and the synchronous `raise()` function to the asynchronous `kill()` function.

LOCALETYPE

Specifies the type of locale support to be used by the object that is created.

LOCALETYPE Syntax



Notes:

[1] C compiler only

***LOCALE**
Default setting. Objects compiled with this option use the locale support provided with the ILE C/C++ compiler and runtime, using locale objects of type *LOCALE. This option is only valid for programs that run on V3R7 and later releases of the IBM i operating system.

***LOCALEUCS2**
Objects compiled with this option store wide-character literals in two-byte form in the UNICODE CCSID (13488).

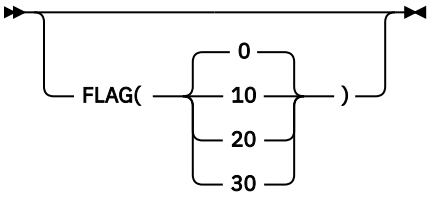
***CLD** C
Objects compiled with this option use the locale support provided with earlier releases of the ILE C compiler and runtime, using locale objects of type *CLD.

***LOCALEUTF**
Module and program objects created with this option use the locale support provided by *LOCALE objects. Wide-character types will contain four-byte utf-32 values. Narrow character types will contain utf-8 values.

FLAG

Specifies the level of messages that are to be displayed in the listing. Only the first-level text of the message is included unless OPTION(*SECLVL) is specified.

FLAG Syntax



0
Default setting. All messages starting at the **informational** level are displayed.

10
All messages starting at the **warning** level are displayed.

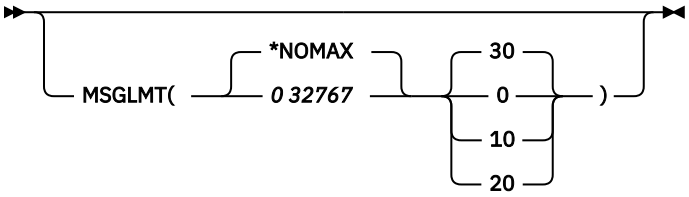
20
All messages starting at the **error** level are displayed.

30
All messages starting at the **severe error** level are displayed.

MSGLMT

Specifies the maximum number of messages at a given severity that can occur before compilation stops.

MSGLMT Syntax



***NOMAX**
Default setting. Compilation continues regardless of the number of messages that have occurred at the specified message severity level.

0 32767
Specifies the maximum number of messages that can occur at, or above, the specified message severity level before compilation stops. The valid range is 0 to 32 767.

30
Default setting. Specifies that *message-limit* messages at severity 30 can occur before compilation stops.

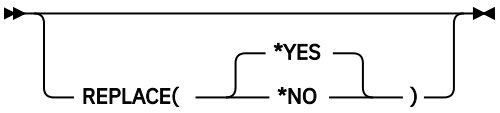
0
Specifies that *message-limit* messages at severity 0 to 30 can occur before compilation stops.

10
Specifies that *message-limit* messages at severity 10 to 30 can occur before compilation stops.

20
Specifies that *message-limit* messages at severity 20 to 30 can occur before compilation stops.

REPLACE

Specifies whether the existing version of the object is to be replaced by the current version.
REPLACE Syntax

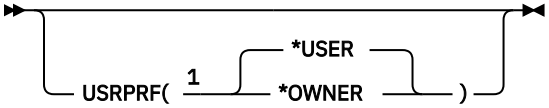


***YES**
Default setting. The existing object is replaced by the new version. The old version is moved to the library, QRPLOBJ, and renamed based on the system date and time. The text description of the replaced object is changed to the name of the original object. The old object is deleted at the next IPL if it has not been deleted.

***NO**
The existing object is not replaced. When an object with the same name exists in the specified library, a message is displayed and compilation stops.

USRPRF

Valid only with the CRTBNDC and CRTBNDCPP commands. Specifies the user profile that is used when the compiled ILE C or C++ program object is run, including the authority that the program object has for each object. The profile of either the program owner or the program user is used to control which objects are used by the program object.
USRPRF Syntax



Notes:

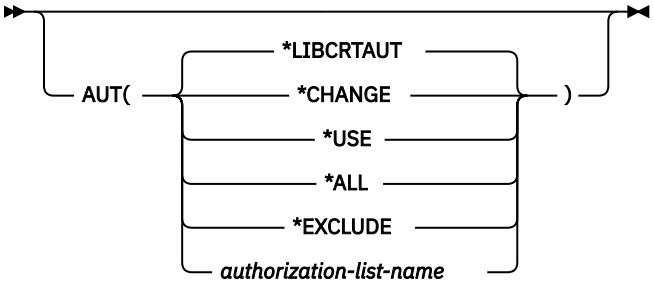
[1] Create Bound Program command only

***USER**
Default setting. The profile of the user that is running the program object is used.

***OWNER**
The collective sets of object authority in the user profiles of both the program owner and the program user are used to find and access objects during the program object's processing time. Objects that are created by the program are owned by the program's user.

AUT

Specifies the object authority to users who do not have specific authority to the object. The user may not be on the authorization list, or whose group has no specific authority to the object.
AUT Syntax



***LIBCRTAUT**
Default setting. Public authority for the object is taken from the CRTAUT keyword of the target library (the library that contains the created object). This value is determined when the object is created. If the CRTAUT value for the library changes after the object is created, the new value does not affect any existing objects.

***CHANGE**
Provides all data authority and the authority to perform all operations on the object except those that are limited to the owner or controlled by object authority and object management authority. The object can be changed, and basic functions can be performed on it.

***USE**
Provides object operational authority, read authority, and authority for basic operations on the object. Users without specific authority are prevented from changing the object.

***ALL**
Provides authority for all operations on the object except those that are limited to the owner or controlled by authorization list management authority. Any user can control the object's existence, specify its security, and perform basic functions on it, but cannot transfer its ownership.

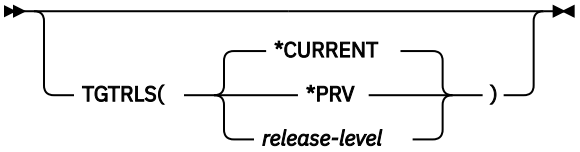
***EXCLUDE**
Users without special authority cannot access the object.

authorization-list-name
Enter the name of an authorization list of users and authorities to which the module object is added. The object is secured by this authorization list, and the public authority for the object is set to *AUTL. The authorization list must exist on the system when the command is issued.

TGTRLS

Specifies the release level of the operating system for the object that is being created.

TGTRLS Syntax



***CURRENT**
Default setting. The object is used on the release of the operating system that is running on your system. For example, when V2R3M5 is running on your system, *CURRENT indicates you want to use the object on a system with Version 2 Release 3 Modification 5 installed. You can also use the object on a system with any subsequent release of the operating system that is installed.

Note : If V2R3M5 is running on your system, and you intend to use the object you are creating on a system with V2R3M0 installed, specify TGTRLS(V2R3M0), not TGTRLS(*CURRENT).

***PRV**
The object is used on the previous release of the operating system. For example, if V2R3M5 is being run on the your system, specify *PRV if you want to use the object you are creating on a system with

V2R2M0 installed. You can also use the object on a system with any subsequent release of the operating system that is installed.

release-level
Specify the release in the format VxRxMx. The object can be used on a system with the specific release or with any subsequent release of the installed operating system. Values depend on the current version, release, and modification level, and they change with each new release. If you specify a release-level that is earlier than the earliest release level supported by this command, you will receive an error message indicating the earliest supported release.

Compiling for an operating system release earlier than V5R1M0 may cause some settings of the following compiler options to be ignored:

- "CHECKOUT" on page 73
- "OPTION" on page 68
- "OUTPUT" on page 68
- "PRFDTA" on page 88

The following options are ignored completely when compiling for an operating system release earlier than V5R1M0:

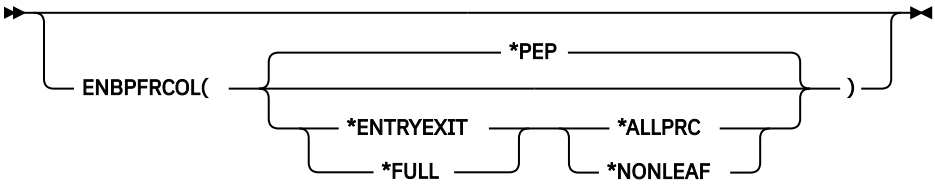
- "CSOPT" on page 97
- "DFTCHAR" on page 97
- "DTAMDL" on page 92
- "ENUM" on page 93
- "INCDIR" on page 96
- "LICOPT" on page 97
- "MAKEDEP" on page 94
- "PACKSTRUCT" on page 92
- "PPGENOPT" on page 94
- "STGMDL" on page 91
- "TGTCCSID" on page 98

ENBPFRCOL

Specifies whether performance data measurement code should be generated in the object. The collected data can be used by the system performance tool to profile performance of an application. Generating performance measurement code in a created object results in slightly larger objects and might affect performance.

Note : Starting in V6R1M0, this parameter no longer affects the created objects. It exists solely for compatibility with releases earlier than V6R1M0.

ENBPFRCOL Syntax



***PEP**
Default setting. Performance statistics are gathered on the entry and exit of the program entry procedure only. Choose this value when you want to gather overall performance information for an application. This support is equivalent to the support that was formerly provided with the TPST tool.

***ENTRYEXIT *NONLEAF**
Performance statistics are gathered on the entry and exit of all the program's procedures that are not leaf procedures. This includes the program PEP.

This choice is useful if you want to capture information about functions that call other functions in your application.

***ENTRYEXIT *ALLPRC**
Performance statistics are gathered on the entry and exit of all the object's procedures (including those that are leaf procedures). This includes the program PEP.

This choice is useful if you want to capture information about all functions. Use this option when you know that all the programs called by your application were compiled with either the *PEP, *ENTRYEXIT, or *FULL option. Otherwise, if your application calls other objects that are not enabled for performance measurement, the performance tool charges their use of resources against your application. This makes it difficult to determine where resources are actually being used.

***FULL *NONLEAF**
Performance statistics are gathered on entry and exit of all procedures that are not leaf procedures. Statistics are gathered before and after each call to an external procedure.

***FULL *ALLPRC**
Performance statistics are gathered on the entry and exit of all procedures that include leaf procedures. Also, statistics are gathered before and after each call to an external procedure.

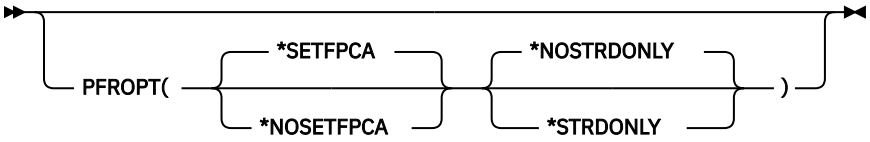
Use this option if your application calls other objects that were not compiled with either the *PEP, *ENTRYEXIT, or *FULL option. This option allows the performance tools to distinguish between resources that are used by your application and those resources used by objects it calls (even if those objects are not enabled for performance measurement). This option is the most expensive, but allows for selectively analyzing various programs in an application.

***NONE**
No performance data is collected for this object. Use this parameter when no performance information is needed, and a smaller object size is required.

PFROPT

Specifies various options available to boost performance. You can specify them in any order, separated by one or more blanks. When an option is specified more than once, or when two options conflict, the last option specified is used.

PFROPT Syntax



***SETFPCA**
Default setting. Causes the compiler to set the floating-point computational attributes to achieve the ANSI semantics for floating-point computations.

***NOSETFPCA**
No computational attributes will be set. This option is used when the object being created does not have any floating-point computations in it.

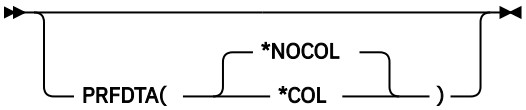
***NOSTRDONLY**
Specifies that the compiler must place strings into writable memory. This is the default.

***STRDONLY**
Specifies that the compiler may place strings into read-only memory.

PRFDTA

Specifies whether program profiling should be turned on for the module or program. Profiling can lead to better performance of your programs or service programs by improving the use of cache lines and memory pages in ILE applications.

PRFDTA Syntax



Note : You cannot profile a stand-alone *MODULE object.

***NOCOL**

Default setting. The collection of profiling data is not enabled. The module will not collect profiling data when it is included in a program or service program object.

***COL**

The collection of profiling data is enabled. The module will collect profiling data when it is included in a program or service program object.

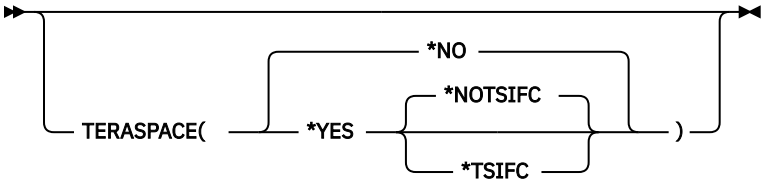
Use this option to generate code that will collect data at object creation time. This data will consist of the number of times basic blocks within procedures are executed, as well as the number of times procedures are called.

Note : *COL has an effect only when the optimization level of the module is *FULL (30) or greater.

TERASPACE

Specifies whether the created object can recognize and work with addresses that reference teraspace storage locations.

TERASPACE Syntax



***NO**

Default setting. The created object cannot recognize teraspace storage addresses.

Note : Starting in V6R1M0, all modules are enabled to handle addressing of storage allocated from teraspace. However, if *NO is specified, the compiler facilities listed in the *YES description are not available.

***YES**

The created object can handle teraspace storage addresses, including parameters passed from other teraspace-enabled programs and service programs. In addition, the following compiler facilities are enabled:

- Pointers can be qualified with __ptr64 to allow creation of 8-byte pointers used to access teraspace storage.
- The teraspace storage model can be specified with the STGMDL(*TERASPACE) compiler option.
- The LLP64 data model can be specified with the DTAMDL(*LLP64) compiler option or the `#pragma datamodel(llp64)` directive.
- Pointer difference arithmetic returns a signed long long result instead of a ptrdiff_t result.

***NOTSIFC**

The compiler does not use teraspace versions of storage functions, such as malloc( ) or shmat( ). *NOTSIFC is the default if TERASPACE(*YES) is specified.

***TSIFC**
The compiler will use teraspace versions of storage functions, such as malloc( ) or shmat( ), without requiring changes to the program source code. The compiler defines the __TERASPACE__ macro, and maps certain storage function names to their teraspace-enabled equivalents. For example, selecting this compiler option causes the malloc( ) storage function to be mapped to _C_TS_malloc( ).

The DTAMDL (see page "DTAMDL" on page 92) and STGMDL (see page "STGMDL" on page 91) compiler options can be used together with the TERASPACE compiler option. Valid combinations of these options are shown in the following tables, along with the effects of selecting those combinations.

Table 4. Valid Combinations of DTAMDL, STGMDL, and TERASPACE Compiler Options

DTAMDL(*P128)	**STGMDL**		
	(*SNGLVL)	**(*TERASPACE)**	**(*INHERIT)**
	• Module/program is designed to use single-level store working storage. • Generated code supports execution using: – single-level store working storage – single-level store dynamic storage • Working storage can only be accessed using 16–byte space pointers. • Default pointer size is 16 bytes.	• Module/program is designed to use teraspace working storage. • Generated code supports execution using: – teraspace working storage – single-level store dynamic storage – teraspace dynamic storage • Working storage can be accessed using either: – process local pointers – 16–byte space pointers • Default pointer size is 16 bytes.	• Depending on the storage model of the calling program, the module is designed to use either: – single-level store working storage – teraspace working storage • Depending on the storage model of the calling program, generated code supports execution using: – single-level store working storage – teraspace working storage – single-level store dynamic storage – teraspace dynamic storage • Default pointer size is 16 bytes.
TERASPACE(*NO)	Default setting	Invalid combination	Invalid combination
TERASPACE(*YES *NOTSIFC)	• Generated code also supports execution using teraspace • Default is to use single-level store version of dynamic storage interfaces.	• Default is to use single-level store version of dynamic storage interfaces.	• Default is to use single-level store version of dynamic storage interfaces.

Table 4. Valid Combinations of DTAMDL, STGMDL, and TERASPACE Compiler Options (continued)

DTAMDL(*P128)	STGMDL		
	(*SNGLVL)	**(*TERASPACE)**	**(*INHERIT)**
TERASPACE(*YES *TSIFC)	• Generated code also supports execution using teraspace. • Default is to use teraspace version of dynamic storage interfaces. • __TERASPACE__ macro is defined.	• Default is to use teraspace version of dynamic storage interfaces. • __TERASPACE__ macro is defined.	• Default is to use teraspace version of dynamic storage interfaces. • __TERASPACE__ macro is defined.

DTAMDL(*LLP64)	STGMDL		
	(*SNGLVL)	**(*TERASPACE)**	**(*INHERIT)**
	• Module/program is designed to use single-level store working storage. • Generated code supports execution using: – single-level store working storage – single-level store dynamic storage – teraspace • Working storage can only be accessed using 16–byte space pointers. • Default pointer size is 8 bytes.	• Module/program is designed to use teraspace working storage. • Generated code supports execution using: – teraspace working storage – single-level store dynamic storage – teraspace dynamic storage • Working storage can be accessed using either: – process local pointers – 16–byte space pointers • Default pointer size is 8 bytes.	• Depending on the storage model of the calling program, the module is designed to use either: – single-level store working storage – teraspace working storage • Depending on the storage model of the calling program, generated code supports execution using: – single-level store working storage – teraspace working storage – single-level store dynamic storage – teraspace dynamic storage • Working storage can be accessed using either: – (conditionally) process local pointers – 16–byte space pointers • Default pointer size is 8 bytes.
TERASPACE(*NO)	Invalid combination	Invalid combination	Invalid combination

DTAMDL(*LLP64)	STGMDL		
	(*SNGLVL)	(*TERASPACE)	(*INHERIT)
TERASPACE(*YES *NOTSIFC)	• Default is to use single-level storage version of dynamic storage interfaces. • __LLP64_IFC__ macro is defined.	• Default is to use single-level storage version of dynamic storage interfaces. • __LLP64_IFC__ macro is defined.	• Default is to use single-level storage version of dynamic storage interfaces. • __LLP64_IFC__ macro is defined.
TERASPACE(*YES *TSIFC)	• Default is to use teraspace version of dynamic storage interfaces. • __TERASPACE__ and __LLP64_IFC__ macros are defined.	***Recommended settings for most effective use of teraspace*** • Default is to use teraspace version of dynamic storage interfaces. • __TERASPACE__ and __LLP64_IFC__ macros are defined.	• Default is to use teraspace version of dynamic storage interfaces. • __TERASPACE__ and __LLP64_IFC__ macros are defined.

To make the most effective use of teraspace, you can specify the following combination of options:

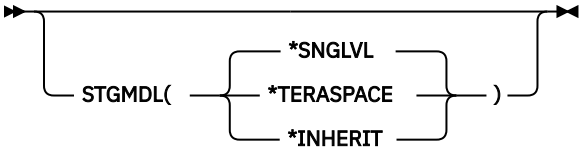
```
TERASPACE(*YES *TSIFC) STGMDL(*TERASPACE) DTAMDL(*LLP64)
```

For more information about teraspace storage, see *Using Teraspace* in the *ILE C/C++ Programmer's Guide* and *Teraspace and single-level store* in the *ILE Concepts*.

STGMDL

Specifies the type of storage (static and automatic) that the module object uses.

STGMDL Syntax



***SNGLVL**

Default setting. The module or program uses the traditional single level storage model. Static and automatic storage for the object is allocated from single-level store, and can only be accessed using 16-byte pointers. The module can optionally access teraspace dynamic storage if the TERASPACE(*YES) option is specified.

***TERASPACE**

The module or program uses the teraspace storage model. Teraspace storage model provides up to a 1-terabyte local address space for a single job. Static and automatic storage for the object is allocated from teraspace and can be accessed using either 8-byte or 16-byte pointers.

***INHERIT**

Valid only with the CRTCMOD and CRTCPPMOD commands. The module created can use either single level or teraspace storage. The type of storage used depends on the type of storage required by the caller.

Use of STGMDL(*TERASPACE) or STGMDL(*INHERIT) together with TERASPACE(*NO) is flagged as an error by the compiler, and compilation stops.

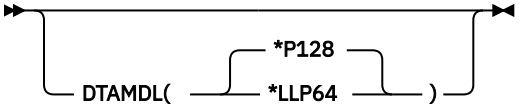
For more information about valid combinations for the STGMDL, TERASPACE, and DTAMDL compiler options, see "TERASPACE" on page 88.

For more information about the types of storage available on IBM i platforms, see *Teraspace and single-level store* in *ILE Concepts*.

DTAMDL

Specifies how pointer types are interpreted in absence of an explicit modifier. The __ptr64 and __ptr128 type modifiers and the datamodel pragma override the setting of the DTAMDL compiler option.

DTAMDL Syntax



***P128**
Default setting. The default size of pointer variables is 16 bytes.

***LLP64**
The default size of pointer variables is 8 bytes, and the compiler defines the macro __LLP64_IFC__.

Use of DTAMDL(*LLP64) together with TERASPACE(*NO) is flagged as an error by the compiler, and compilation stops.

See pragma "datamodel" on page 27 for more information.

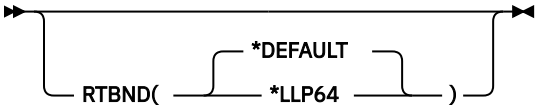
For more information about valid combinations for the STGMDL, TERASPACE, and DTAMDL compiler options, see "TERASPACE" on page 88.

RTBND

C++

Specifies the runtime binding directory for the object created.

RTBND Syntax



***DEFAULT**
Default setting. The object created uses the default binding directory.

***LLP64**
The object created uses the 64-bit runtime binding directory and the compiler defines the macro __LLP64_RTBND__.

PACKSTRUCT

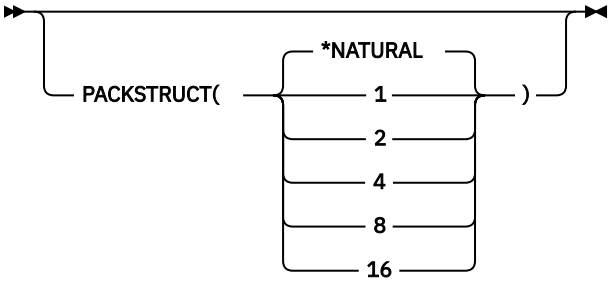
Specifies the alignment rules to use for members of structures, unions, and classes in the source code. PACKSTRUCT sets the packing value to be used for the members of structures and for the structures themselves.

If the data types are by default packed along boundaries smaller than those boundaries specified by #pragma pack, they are still aligned along the smaller boundaries. For example:

- Type char is always aligned along a 1-byte boundary.
- 16-byte pointers are aligned on a 16-byte boundary. PACKSTRUCT, _Packed, and #pragma pack cannot alter this alignment.
- 8-byte pointers can have any alignment, but 8-byte alignment is preferred.

For more information about packing and alignment, see pragma "pack" on page 51.

PACKSTRUCT Syntax



***NATURAL**
Default setting. The natural alignment for the members of structures is used.

1
Structures and unions are packed along 1-byte boundaries.

2
Structures and unions are packed along 2-byte boundaries.

4
Structures and unions are packed along 4-byte boundaries.

8
Structures and unions are packed along 8-byte boundaries.

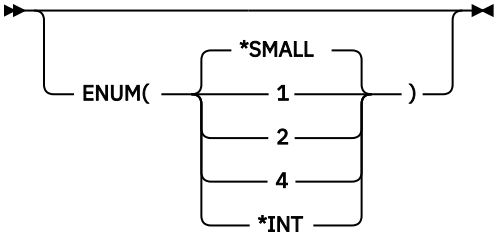
16
Structures and unions are packed along 16-byte boundaries.

ENUM

Specifies the number of bytes the compiler uses to represent enumerations. This becomes the default enumeration size for the object. A #pragma enum directive overrides this compile option.

C++11 The **ENUM** option affects only unscoped enumerations that have no fixed underlying type. For enumerations with a fixed underlying type, the **ENUM** option is ignored.

ENUM Syntax



***SMALL**
Default setting. Use the smallest possible size for an enum, as appropriate to the given enum value.

1
Make all enum variables 1 byte in size, signed if possible

2
Make all enum variables 2 bytes in size, signed if possible

4
Make all enum variables 4 bytes in size, signed if possible

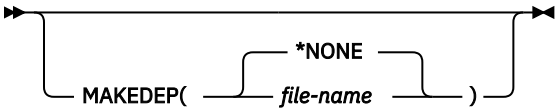
***INT**

- C Use the ANSI C Standard enum size (4-bytes signed).
- C++ Use the ANSI C++ Standard enum size (4-bytes signed; unless the enumeration value > 2^{31}-1).

MAKEDEP

Creates an output file containing targets suitable for inclusion in a description file for the Qshell make command.

MAKEDEP Syntax



*NONE
Default setting. The option is disabled and no file is created.

file-name
Specifies an IFS path indicating the location and name of the resulting output file.

The output file contains a line for the input file and an entry for each include file. It has the general form:

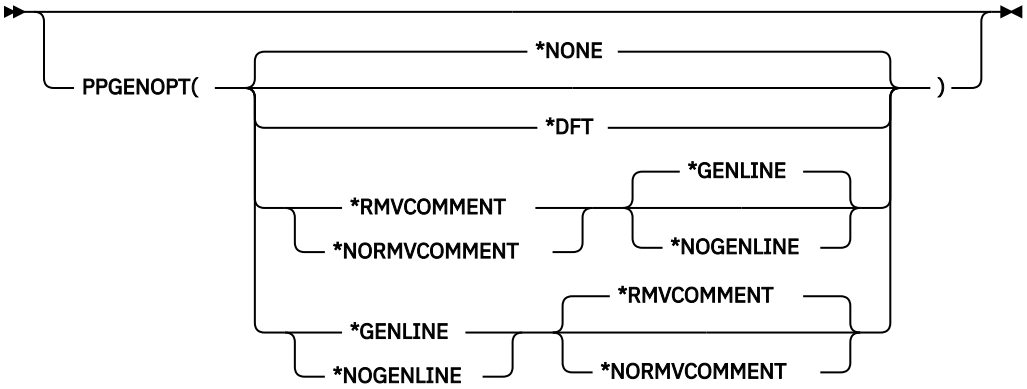
```
file_name.o:file_name.c
file_name.o:include_file_name
```

Include files are listed according to the search order rules for the #include preprocessor directive. If an include file is not found, it is not added to the output file. Files with no include statements produce output files containing one line that lists only the input file name.

PPGENOPT

Valid only with the CRTCMOD or CRTCPPMOD commands. Lets you specify outputs generated by the preprocessor.

PPGENOPT Syntax



*NONE
Default setting. No outputs are generated by the preprocessor. Selecting this option disables the PPSRCFILE, PPSRCMBR, and PPSRCSTMF options.

***DFT**
Equivalent to specifying PPGENOPT(*RMVCOMMENT *GENLINE).

***RMVCOMMENT**
Preserves comments during preprocessing.

***NORMVCOMMENT**
Does not preserve comments during preprocessing.

***NOGENLINE**
Suppresses #line directives in the preprocessor output.

***GENLINE**
Produces #line directives in the preprocessor output.

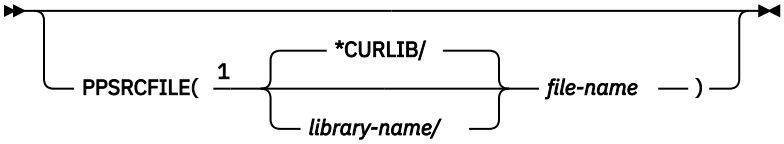
Notes:

1. Specifying the PPGENOPT compiler option with any setting other than *NONE forces the input of either of the following options:
   - PPSRCFILE and PPSRCMBR
   - PPSRCSTMF
2. C Specifying PPGENOPT with any setting other than *NONE overrides the OPTION(*NOPPONLY) and OPTION(*GEN) option settings.
3. C Specifying OPTION(*PPONLY) overrides the PPGENOPT(*NONE) and OPTION(*GEN) option settings. Instead, the following settings are implied:
   - PPGENOPT(*DFT) PPSRCFILE(QTEMP/QACZEXPAND) PPSRCMBR(*MODULE) for a data management source file.
   - PPGENOPT(*DFT) PPSRCSTMF(*SRCSTMF) for an IFS source file.

PPSRCFILE

Valid only with the CRTCMOD or CRTCPPMOD commands. This option is used together with the PPGENOPT option to define where the preprocessor output object is stored.

PPSRCFILE Syntax



Notes:

1 Create Module command only

***CURLIB**
Default setting. The object is stored in the current library. If a job does not have a current library, QGPL is used.

library-name
The name of the library where the preprocessor output is stored.

file-name
The physical file name under which the preprocessor output is stored. The file is created if it does not already exist.

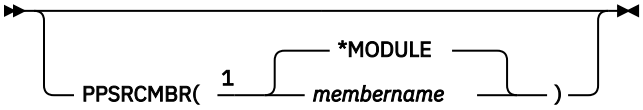
Notes:

1. The PPSRCMBR and PPSRCFILE options cannot be specified with the PPSRCSTMF option.
2. C Specifying OPTION(*PPONLY) for a data management file implies the following settings:
   - PPGENOPT(*DFT) PPSRCFILE(QTEMP/QACZEXPAND) PPSRCMBR(*MODULE)

PPSRCMBR

Valid only with the CRTCMOD or CRTCPPMOD commands. This option is used together with the PPGENOPT option to define the name of the member where preprocessor output is stored.

PPSRCMBR Syntax



Notes:

1 Create Module command only

***MODULE**
The module name that is supplied on the MODULE parameter is used as the source member name. This is the default when a member name is not specified.

member-name
Enter the name of the member that will contain the preprocessor output.

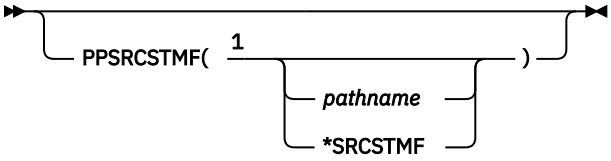
Notes:

1. The PPSRCMBR and PPSRCFILE options cannot be specified with the PPSRCSTMF option.
2. C Specifying OPTION(*PPONLY) for a data management file implies the following settings:
   - PPGENOPT(*DFT) PPSRCFILE(QTEMP/QACZEXPAND) PPSRCMBR(*MODULE)

PPSRCSTMF

Valid only with the CRTCMOD or CRTCPPMOD commands. This option is used together with the PPGENOPT option to define the IFS stream path name where preprocessor output is stored.

PPSRCSTMF Syntax



Notes:

1 Create Module command only

path-name
Enter the IFS path of the file that will contain the preprocessor output. The path name can be either absolutely or relatively qualified. An absolute path name starts with '/'; a relative path name starts with a character other than '/'. If absolutely qualified, then the path name is complete. If relatively qualified, the path name is completed by pre-pending the job's current working directory to the path name.

***SRCSTMF**
If this setting is chosen, you must also select the SRCSTMF command option. Preprocessor output is saved to the current directory under the same base filename specified by the SRCSTMF command option, but with a filename extension of **.i**.

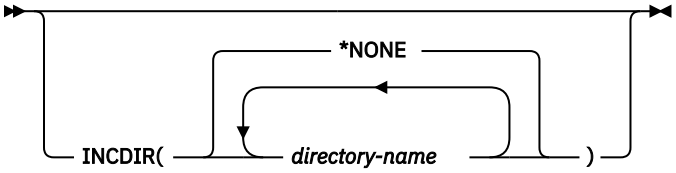
Notes:

1. The PPSRCMBR and PPSRCFILE options cannot be specified with the PPSRCSTMF option.
2. The SRCSTMF parameter is not supported in a mixed-byte environment.
3. C Specifying OPTION(*PPONLY) for an IFS file implies the following settings:
   - PPGENOPT(*DFT) PPSRCSTMF(*SRCSTMF)

INCDIR

Lets you redefine the path used to locate include header files when compiling a source stream file. This option is ignored if the source file's location is not defined as an IFS path with the SRCSTMF compiler option, or if the full absolute path name is specified on the #include directive.

INCDIR Syntax



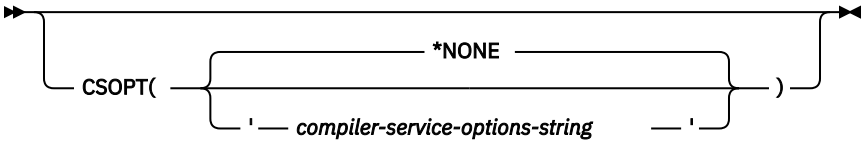
***NONE**
Default setting. No directories are inserted at the start of the default user include path.

directory-name
Specifies a directory name to be inserted at the start of the default user include path. More than one directory name can be entered. Directories are inserted at the start of the default user include path in the order they are entered.

CSOPT

This option lets you specify one or more compiler service options. Valid option strings will be described in PTF cover letters and release notes.

CSOPT Syntax



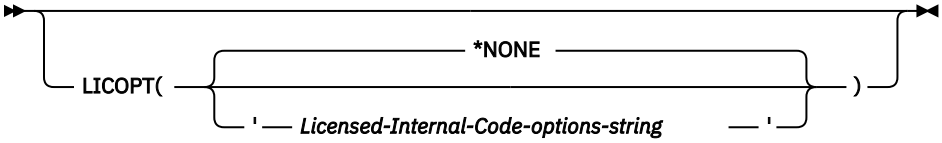
***NONE**
Default setting. No compiler service options selected.

compiler-service-options-string
Specified compiler service options are used when creating a module object.

LICOPT

Specifies one or more Licensed Internal Code compile time options. This parameter allows individual compile time options to be selected, and is intended for the advanced programmer who understands the potential benefits and drawbacks of each selected type of compiler option.

LICOPT Syntax



The possible options are:

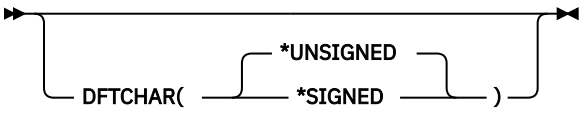
***NONE**
Default setting. No compile time optimization is selected.

Licensed-Internal-Code-options-string
The selected Licensed Internal Code compile time options are used when creating the module/program object. Certain options may reduce your ability to debug the created module/program. See *ILE Concepts* for more information about LICOPT options.

DFTCHAR

Instructs the compiler to treat all variables of type **char** as either signed or unsigned.

DFTCHAR Syntax



***UNSIGNED**
Default setting. Treats all variables declared as type **char** as type **unsigned char**. The _CHAR_UNSIGNED macro is defined.

***SIGNED**
Treats all variables declared as type **char** as type **signed char**, and defines the _CHAR_SIGNED macro. This setting is ignored if the TGTRLS option specifies a target release earlier than V5R1M0.

TGTCCSID

Specifies the target coded character set identifier (CCSID) of the created object. The object's CCSID identifies the coded character set identifier in which the module's character data is stored. This includes character data used to describe literals, comments and identifier names described by the source, with the exception of identifier names for CCSIDs 5026, 930 and 290.

C

If an ASCII CCSID is entered, the compiler issues an error message and assumes a CCSID of 37.

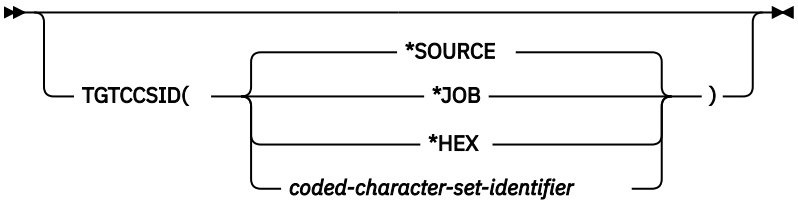
C++

If an ASCII CCSID is entered, the compiler issues no error message. Translation occurs to the ASCII CCSID but the created module has a CCSID of 65535.

The TGTCCSID option will also determine the CCSID of character values used in listings. However, listings sent to a spool file will be in the job's CCSID because that is the CCSID of the spool file.

This option is ignored when targeting a compile for a release previous to V5R1.

TGTCCSID Syntax



***SOURCE**
Default setting. The CCSID of the root source file is used.

***JOB**
The CCSID of the current job is used.

***HEX**
The CCSID 65535 is used, indicating that character data is treated as bit data and is not converted.

coded-character-set-identifier
Specifies a specific CCSID to be used.

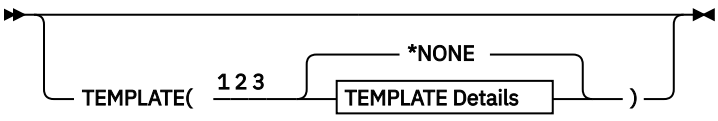
Note : When *HEX (CCSID 65535) is specified, target CCSID will be used in this order: source CCSID, job CCSID, then system QCCSID value. If they are all 65535, target CCSID 37 will be used.

TEMPLATE

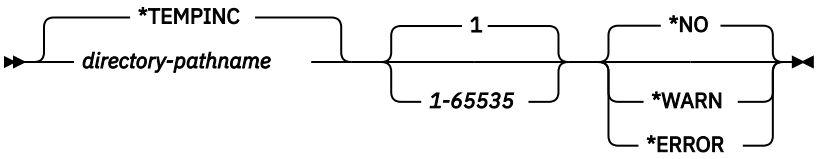
C++

Specifies options to customize C++ template generation.

TEMPLATE Syntax



TEMPLATE Details



Notes:

[1] C++ compiler only
[2] Create Module command only
[3] Applicable only when using the Integrated File System (IFS)

The possible options are:

***NONE**
No automatic template instantiation file is created. The compiler instantiates all templates whose full implementation is known if an object of that template class is defined, or if a call is made to that template function within the module. If the full implementation is not known (for example, you have a template class definition, but not the definition of the methods of that template class), that template is not instantiated within the module.

Note : This can cause code duplication in program executables where template specifications are used in more than one module.

***TEMPINC**
Templates are generated into a directory named **tempinc** which is created in the directory where the root source file was found. If the source file is not a stream file, a file named TEMPINC will be created in the library where the source file resides. The TEMPLATE(*TEMPINC) and TMPLREG options are mutually exclusive.

directory-pathname
Same as *TEMPLATE(*TEMPINC), except that template instantiation files are generated to a specified directory location. The directory path can be relative to the current directory, or it can be an absolute directory path.

If the specified directory does not exist, it is created.

Note :

An error condition results if the specified directory path contains a directory that does not exist, for example, TEMPLATE(/source/subdir1/tempinc) when subdir1 does not exist.

1-65535
Specifies the maximum number of template include files to be generated by the *TEMPLATE(*TEMPINC) option for each header file. If not specified, this setting defaults to **1**. The maximum value for this setting is 65535.

***NO**
Default setting if TEMPLATE(*NONE) is not in effect. If specified, the compiler does not parse to reduce the number of errors issued in code written for previous versions of the compiler.

Note : Regardless of the setting of this and the next two options, error messages are produced for problems that appear outside implementations. For example, errors found during the parsing or semantic checking of constructs such as the following, always cause error messages:

- return type of a function template
- parameter list of a function template
- member list of a class template
- base specifier of a class template

***WARN**
Parses template implementations and issues warning messages for semantic errors. Error messages are also issued for errors found while parsing.

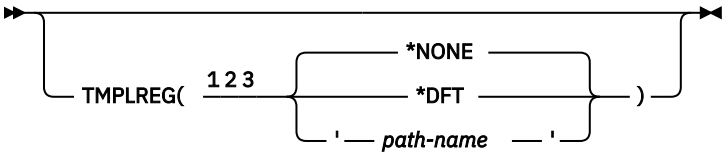
***ERROR**
Treats problems in template implementations as errors, even if the template is not instantiated.

TMPLREG

C++

Valid only with the CRTCPPMOD command. Maintains a record of all templates as they are encountered in the source and ensures that only one instantiation of each template is made. The TMPLREG and TEMPLATE(*TEMPINC) parameters are mutually exclusive.

TMPLREG Syntax



Notes:

1 C++ compiler only
2 Create Module command only
3 Applicable only when using the Integrated File System (IFS)

The possible options are:

***NONE**
Default setting. Do not use the template registry file to keep track of template information.

***DFT**
If the source file is a stream file, the template registry file is created in the source directory with the default name 'templateregistry'. If the source file is not a stream file, a file QTMPLREG with the member QTMPLREG will be created in the library where the source resides.

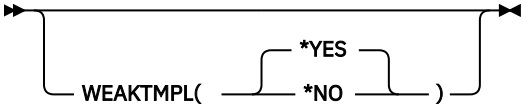
path-name
Specifies a path name for the stream file in which to store the template registry information.

WEAKTMPL

C++

Specifies whether weak definitions are used for static members of a template class. Weakly defined static members of a template class will prevent the collisions of multiple definitions in a program or service program.

WEAKTMPL Syntax



The possible options are:

***YES**
Default setting. Weak definitions will be used for static members of a template class.

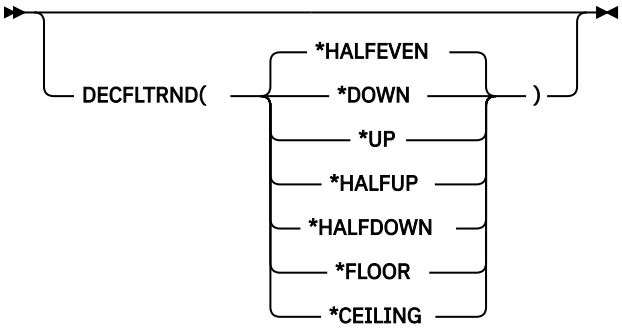
***NO**
Weak definitions will not be used for static members of a template class.

Some programs require strong static data members when they are linked to other modules. You can override the default only at compilation time.

DECFLTRND

Specifies the compile time rounding mode for the evaluation of constant decimal floating-point expressions. This option does not affect the runtime decimal floating-point rounding mode, which is set using the setca built-in function.

DECFLTRND Syntax



The possible options are:

***HALFEVEN**
Default setting. Round to the nearest value. In a tie, choose even. For example, 5.22 rounds to 5.2, 5.67 rounds to 5.7, 5.55 rounds to 5.6, 5.65 rounds to 5.6.

***DOWN**
Round toward zero, or truncate the result. For example, 5.22 rounds to 5.2, 5.67 rounds to 5.6, 5.55 rounds to 5.5, 5.65 rounds to 5.6

***UP**
Round away from zero. For example, 5.22 rounds to 5.3, 5.67 rounds to 5.7, 5.55 rounds to 5.6, 5.65 rounds to 5.7.

***HALFUP**
Round to the nearest value. In a tie, round away from zero. For example, 5.22 rounds to 5.2, 5.67 rounds to 5.7, 5.55 rounds to 5.6, 5.65 rounds to 5.7.

***HALFDOWN**
Round to the nearest value. In a tie, round toward zero. For example, 5.22 rounds to 5.2, 5.67 rounds to 5.7, 5.55 rounds to 5.5, 5.65 rounds to 5.6.

***FLOOR**
Round toward negative affinity. For example, 5.22 rounds to 5.2, 5.67 rounds to 5.6, 5.55 rounds to 5.5, 5.65 rounds to 5.6

***CEILING**
Round toward positive infinity. For example, 5.22 rounds to 5.3, 5.67 rounds to 5.7, 5.55 rounds to 5.6, 5.65 rounds to 5.7.

Using the ixlc Command to Invoke the C/C++ Compiler

Read this section for an overview of the ixlc Qshell command.

The `ixlc` command lets you invoke the compiler and specify compiler options from a IBM i Qshell command line. Module binder commands can be specified. The ixlc command can be used together with AIX make files to control compilation.

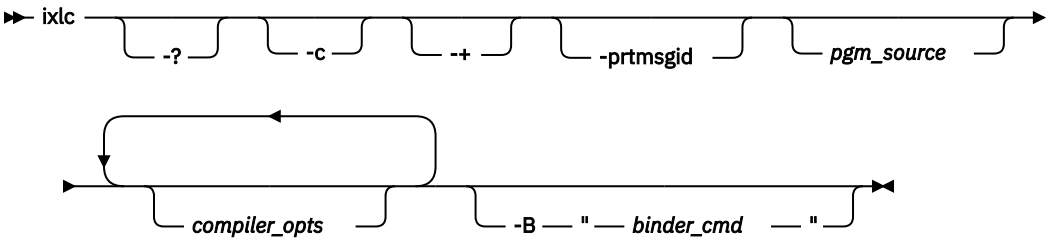
Using ixlc in Qshell

When using the IBM i version of `ixlc` on the character-based interface Qshell command line, you can:

- Compile data management source code residing on a IBM i platform.
- Compile IFS source code residing on a IBM i platform.
- Use header files residing on the IBM i platform.

ixlc Command and Options Syntax

Basic syntax for the ixlc command is:



where:

ixlc
Basic compiler command invocation. By default, the `ixlc` command instructs the compiler to create a bound program.

-?
Specifying this flag displays help for the ixlc command.

-c
Specifying this flag instructs the compiler to create a module.

-+
Specifying this flag invokes the C++ compiler.

-prtmsgid
Specifying this flag causes additional information on compiler error messages to be displayed. The additional information includes the line number, column number, message identifier, and message severity.

pgm_source
Specifies the name of the program source file being compiled. You can compile an IFS source program or data management source program by providing the source name as:

```
qsys.lib/.../name.mbr
```

Alternately, you can also compile a data management source program by using the `-qsrcfile(`*`library/file`*`)` and `-qsrcmbr(`*`member`*`)` Qshell compiler options to identify the location of the program source.

compiler_opts
Specifies the ixlc name of an ILE C/C++ compiler option.

-B"*binder_cmd*"
Specifies a binder command and options. For example:

```
-B"CRTPGM PGM(library/target) MODULE(...)"
```

Notes on Usage

1. ixlc commands and options are case sensitive.
2. It is possible to specify conflicting options when invoking the compiler. If this occurs, options specified later on the command line will override options specified earlier. For example, invoking the compiler by specifying :

   ```
   ixlc hello.c -qgen -qnogen
   ```

   is equivalent to specifying:

   ```
   ixlc hello.c -qnogen
   ```

3. Some option settings are cumulative, and can be specified more than once on the command line without cancelling out earlier specifications of that same option. These options include:
   - settings within the OPTION compiler option group
   - settings within the CHECKOUT compiler option group
   - ALIAS compiler option
   - DEFINE compiler option
   - PPGENOPT compiler option

ixlc Command Options

The table below shows the mappings of Create Module and Create Bound Program compiler options to their `ixlc` equivalents. Compiler options may have language and usage restrictions that are not shown in this table. For information on such restrictions, refer to the reference information for that option.

Table 5. ixlc Command Options

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
"MODULE" on page 65, "PGM" on page 65	[*CURLIB/ | *libraryname*/]*name*	-o[*CURLIB/ | *libraryname*/]*name*
	If library is not specified, the target object goes to the current library as specified by the current user profile. If the user does not have a current library, QGPL is assumed.	
"SRCFILE" on page 66	[*LIBL/ | *CURLIB/ | *libraryname*/] *filename*	-qsrcfile=[*LIBL/ | *CURLIB/ | *libraryname*/] *filename*
"SRCMBR" on page 66	*MODULE | *mbrname*	-qsrcmbr=*mbrname*
"SRCSTMF" on page 67	*pathname*	(*none, uses default pathname*)
"TEXT" on page 67	*SRCMBRTEXT | *BLANK | *text*	-qtext="*text*"
"OUTPUT" on page 68	*NONE	-qnoprint
	*PRINT	-qprint
	filename	-qoutput="*filename*"

Table 5. ixlc Command Options (continued)		
Create Module/Create Bound Program Options	**Option Settings**	**ixlc Equivalents and Notes**
"OPTION" on page 68	*AGR | *NOAGR	-qagr
	*BITSIGN | *NOBITSIGN	-qbitfields=signed -qbitfields=unsigned
	*DIGRAPH | *NODIGRAPH	-qdigraph -qnodigraph
	*EVENTF | *NOEVENTF	-qeventf -qnoeventf
	*EXPMAC | *NOEXPMAC	-qexpmac -qnoexpmac
	*FULL | *NOFULL	-qfull -qnofull
	*GEN | *NOGEN	-qgen -qnogen
	*INCDIRFIRST | *NOINCDIRFIRST	-qidirfirst
	*LOGMSG | *NOLOGMSG	-qlogmsg -qnologmsg
	*LONGLONG | *NOLONGLONG	-qlonglong -qnolonglong
	*NORTTI | *RTTIALL | *RTTITYPE | *RTTICAST	-qnortti -qrtti=all -qrtti=typeinfo -qrtti=dynamiccast
	*PPONLY | *NOPPONLY	-qpponly
	*SECLVL | *NOSECLVL	-qseclvl -qnoseclvl
	*SHOWINC | *NOSHOWINC	-qshowinc -qnoshowinc
	*SHOWSKP | *NOSHOWSKP	-qshowskp -qnoshowskp
	*SHOWSRC | *NOSHOWSRC	-qsource -qnosource
	*SHOWSYS | *NOSHOWSYS	-qshowsys -qnoshowsys
	*SHOWUSR | *NOSHOWUSR	-qshowusr
	*STDINC | *NOSTDINC	-qstdinc -qnostdinc
	*STDLOGMSG | *NOSTDLOGMSG	-qstdlogmsg -qnostdlogmsg
	*STRUCREF | *NOSTRUCREF	-qrefagr
	*SYSINCPATH | *NOSYSINCPATH	-qsysincpath -qnosysincpath
	*XREF | *NOXREF	-qxref=full -qxref
	*XREFREF | *NOXREFREF	-qattr=full -qattr

Table 5. ixlc Command Options (continued)		
Create Module/Create Bound Program Options	**Option Settings**	**ixlc Equivalents and Notes**
"CHECKOUT" on page 73	*NONE | *USAGE | *ALL	-qinfo=cnd -qinfo=all
	*CLASS | *NOCLASS	-qinfo=cls
	*COND | *NOCOND	-qinfo=cnd
	*CONST | *NOCONST	-qinfo=cns
	*EFFECT | *NOEFFECT	-qinfo=eff
	*ENUM | *NOENUM	-qinfo=enu
	*EXTERN | *NOEXTERN	-qinfo=ext
	*GENERAL | *NOGENERAL	-qinfo=gen
	*GOTO | *NOGOTO	-qinfo=got
	*INIT | *NOINIT	-qinfo=ini
	*LANG | *NOLANG	-qinfo=lan
	*PARM | *NOPARM	-qinfo=par
	*PORT | *NOPORT	-qinfo=por
	*PPCHECK | *NOPPCHECK	-qinfo=ppc
	*PPTRACE | *NOPPTRACE	-qinfo=ppt
	*REACH | *NOREACH	-qinfo=rea
	*TEMP | *NOTEMP	-qinfo=gnr
	*TRUNC | *NOTRUNC	-qinfo=trd
	*UNUSED | *NOUNUSED	-qinfo=use
"OPTIMIZE" on page 76	10 | 20 | 30 | 40	-qoptimize=10 -qoptimize=20 -qoptimize=30 -qoptimize=40 -O
		-O is equivalent to specifying -qoptimize=40

Table 5. ixlc Command Options (continued)

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
"INLINE" on page 76	*OFF	-qnoinline
	*ON *AUTO | *NOAUTO 250 | 1-65535 | *NOLIMIT 2000 | 1-65535 | *NOLIMIT *NO | *YES	-qinline="*opt1 opt2 opt3 opt4*" where: • *opt1* is one of: – auto – noauto • *opt2* is one of: – 250 – *1–65536* – *NOLIMIT • *opt3* is one of: – 2000 – *1–65536* – *NOLIMIT • *opt4* is one of: – norpt – rpt
		One selection from each option group *must* be specified. Selections *must* be separated with a space. For example: `-qinline="auto 400 3000 rpt"`
"MODCRTOPT" on page 78	*KEEPILDATA | *NOKEEPILDATA	-qildta -qnoildta
"DBGVIEW" on page 78	*NONE | *ALL | *STMT | *SOURCE | *LIST	-qdbgview=none -qdbgview=all -qdbgview=stmt -qdbgview=source -qdbgview=list -g
		-g is equivalent to specifying -qdbgview=all
"DBGENCKEY" on page 79	*NONE | *character value*	-qdbgenckey=string
"DEFINE" on page 79	*NONE | *name* | *name=value*	-D*name*
		Defines *name* with a value of 1.

Table 5. ixlc Command Options (continued)		
Create Module/Create Bound Program Options	**Option Settings**	**ixlc Equivalents and Notes**
"LANGLVL" on page 80	*EXTENDED | *ANSI | *LEGACY | *EXTENDED0X	-qlanglvl=extended -qlanglvl=ansi -qlanglvl=compat366 -qlanglvl=extended0x
"ALIAS" on page 81	*ANSI | *NOANSI | *ADDRTAKEN | *NOADDRTAKEN | *ALLPTRS | *NOALLPTRS | *TYPEPTR | *NOTYPEPTR	-qalias=ansi -qalias=noansi -qalias=addrtaken -qalias=noaddrtaken -qalias=allptrs -qalias=noallptrs -qalias=typeptr -qalias=notypeptr
"SYSIFCOPT" on page 82	*NOIFSIO | **IFSIO | *IFS64IO	-qnoifsio -qifsio -qifsio=64
	*ASYNCSIGNAL | *NOASYNCSIGNAL	-qasyncsignal -qnoasyncsignal
"LOCALETYPE" on page 82	*LOCALE | *LOCALEUCS2 | *LOCALEUTF | *CLD	-qlocale=locale -qlocale=localeucs2 -qlocale=localeutf -qlocale=cld
"FLAG" on page 83	0 | 10 | 20 | 30	-qflag=0 -qflag=10 -qflag=20 -qflag=30
"MSGLMT" on page 83	*NOMAX | *0-32767* 30 | 0 | 10 | 20	-qmsglmt="*limit severity*" where: *limit* can be *nomax or any integer from *0-32767*, and *severity* can be any one of 0, 10, 20, or 30. The default is: -qmsglmt="*nomax 30"
"REPLACE" on page 84	*YES | *NO	-qreplace -qnoreplace
"USRPRF" on page 84	*USER | *OWNER	-quser -qowner
"AUT" on page 84	*LIBCRTAUT | *CHANGE | *USE | *ALL | *EXCLUDE	-qaut=libcrtaut -qaut=change -qaut=use -qaut=all -qaut=exclude

Table 5. ixlc Command Options (continued)

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
"TGTRLS" on page 85	*CURRENT | *PRV | *release_lvl*	-qtgtrls=*current -qtgtrls=*prv -qtgtrls=V*x*R*x*M*x*
"ENBPFRCOL" on page 86	*PEP	-qenbpfrcol=pep
	*ENTRYEXIT *NONLEAF	-qenbpfrcol=entryexitnonleaf
	*ENTRYEXIT *ALLPRC	-qenbpfrcol=entryexitallprc
	*FULL *NONLEAF	-qenbpfrcol=fullnonleaf
	*FULL *ALLPRC	-qenbpfrcol=fullallprc
"PFROPT" on page 87	*SETFPCA | *NOSETFPCA	-qsetfpca -qnosetfpca
	*NOSTRDONLY | *STRDONLY	-qnoro -qro
"PRFDTA" on page 88	*NOCOL | *COL	-qnoprofile -qprofile
		-qprfdta=*NOCOL -qprfdta=*COL
"TERASPACE" on page 88	*NO	-qteraspace=no
	*YES *NOTSIFC	-qteraspace=notsifc
	*YES *TSIFC	-qteraspace=tsifc
"STGMDL" on page 91	*SNGLVL | *TERASPACE | *INHERIT	-qstoragemodel=snglvl -qstoragemodel=teraspace -qstoragemodel=inherit
"DTAMDL" on page 92	*P128 | *LLP64	-qdatamodel=P128 -qdatamodel=LLP64
"RTBND" on page 92	*DEFAULT | *LLP64	-qrtbnd -qrtbnd=llp64
"PACKSTRUCT" on page 92	1 | 2 | 4 | 8 | 16 | *NATURAL	-qalign=1 -qalign=2 -qalign=4 -qalign=8 -qalign=16 -qalign=natural

Table 5. ixlc Command Options (continued)		
Create Module/Create Bound Program Options	**Option Settings**	**ixlc Equivalents and Notes**
"ENUM" on page 93	1 | 2 | 4 | *INT | *SMALL	-qenum=1 -qenum=2 -qenum=4 -qenum=int -qenum=small
"MAKEDEP" on page 94	*NODEP | *filename*	-M*makefile*
"PPGENOPT" on page 94	*NONE | *DFT	-P
	*RMVCOMMENT | *NORMVCOMMENT	-qppcomment -qnoppcomment
	*GENLINE | *NOGENLINE	-qppline -qnoppline
"PPSRCFILE" on page 95	*CURLIB/*filename*	-qppsrcfile=*CURLIB/*filename*
	libraryname/filename	-qppsrcfile=*libraryname/filename*
	filename	-qppsrcfile=*filename*
"PPSRCMBR" on page 95	*MODULE | *mbrname*	-qppsrcmbr=*module -qppsrcmbr=*mbrname*
"PPSRCSTMF" on page 96	*pathname* | *SRCSTMF	-qppfile=*filename* -qppfile=*srcstmf
"INCDIR" on page 96	*NONE | *pathname*	-I*pathname*
	When used on the command line, specifies directories on a IBM i platform. Include environment variables are overwritten.	
"CSOPT" on page 97	*string*	-qcsopt=*string*
"LICOPT" on page 97	*NONE | *string*	-qlicopt=*string*
"DFTCHAR" on page 97	*SIGNED | *UNSIGNED	-qchar=signed -qchar=unsigned
"TGTCCSID" on page 98	*SOURCE | *JOB | *HEX | *ccsid#*	-qtgtccsid=source -qtgtccsid=job -qtgtccsid=hex -qtgtccsid=*ccsid#*

Table 5. ixlc Command Options (continued)

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
"TEMPLATE " on page 98	*NONE | *pathname*	-qnotempinc -qtempinc=*pathname*
	1 - 65535	-qtempmax=*1-65535*
	*NO | *WARN | *ERROR	-qtmplparse=no -qtmplparse=warn -qtmplparse=error
"TMPLREG " on page 100	*DFT | *NONE	-qtmplreg -qnotmplreg
"WEAKTMPL " on page 100	*YES | *NO	-qweaktmpl -qnoweaktmpl
"DECFLTRND" on page 101	*HALFEVEN | *DOWN | *UP | *HALFUP | *HALFDOWN | *FLOOR | *CEILING	-ydn -ydz -ydi -ydna -ydnz -ydm -ydp

I/O Considerations

Read this section for an overview of I/O considerations.

This section provides information about:

- Data Management Operations on Record Files
- Data Management Operations on Stream Files
- C Streams and File Types
- DDS-to-C/C++ Data Type Mapping

Data Management Operations on Record Files

For more information about data management operations and ILE C/C++ functions available for record files, see the *Database file management* section in the *Files and file systems* category at the IBM i Information Center web site:

```
http://www.ibm.com/systems/i/infocenter
```

Data Management Operations on Stream Files

To use stream files (type=record) with record I/O functions you must cast the FILE pointer to an RFILE pointer.

For more information about data management operations and ILE C/C++ functions available for stream files, see the *Database file management* section in the *Files and file systems* category at the IBM i Information Center web site:

```
http://www.ibm.com/systems/i/infocenter
```

C Streams and File Types

The following table summarizes which file types are supported as streams.

Table 6. Processing C Stream and File Types

Stream	Database	Diskette	Tape	Printer	Display	ICF	DDM	Save
TEXT	Yes	No	No	Yes	No	No	Yes	No
BINARY: Character at a time	Yes	No	No	Yes	No	No	Yes	No
BINARY: Record at a time	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

DDS-to-C/C++ Data Type Mapping

The following table shows DDS data types and the corresponding ILE C/C++ declarations that are used to map fields from externally described files to your ILE C/C++ program. The ILE C/C++ compiler creates fields in structure definitions based on the DDS data types in the externally described file.

Table 7. DDS-to-C/C++ Data Type Mappings

DDS Data Type	Length	Decimal Position	C/C++ Declaration
Indicator	1	0	char INxx_INyy[n]; for unused indicators xx through yy char INxx; for used indicator xx
A - alphanumeric	1-32766	none	char field[n]; (where n = 1 to 32766)
A - alphanumeric variable length VARLEN keyword	1-32740	none	_Packed struct { short len; char data[n]; } field; where n is the maximum length of field
B - binary	1-4	0	short int field;
B - binary	1-4	1-4	char field[2];
B - binary	5-9	0	int field;
B - binary	5-9	1-9	char field[4];
H - hexadecimal	1	none	char field;
H - hexadecimal	2-32766	none	char field[n]; (where n = 2 to 32766)
H - hexadecimal variable length VARLEN keyword	1-32740	none	_Packed struct { short len; char data[n]; } field; where n is the maximum length of field
G - graphic variable length VARLEN keyword	4-1000	none	_Packed struct { short len; wchar_t data[n]; } field; (where n = 4 to 1000)
P - packed decimal	1-31	0-31	decimal (n,p) where n is length and p is decimal position on option d
S - zoned decimal	1-31	0-31	char field[n]; (where n = 1 to 31)
F - floating point	[1]	[1]	float field;
F - floating point	[1]	[1]	double field;
J - DBCS only	4-32766	none	char field[n]; (where n = 4 to 32766 and n is an even number)
E - DBCS either	4-32766	none	char field[n]; (where n = 4 to 32766 and n is an even number)
O - DBCS open	4-32766	none	char field[n]; (where n = 4 to 32766)
J - DBCS only variable length VARLEN keyword	4-32740	none	_Packed struct { short len; char data[n]; } field; (where n = 4 to 32740 and n is an even number)
E - DBCS either variable length VARLEN keyword	4-32740	none	_Packed struct { short len; char data[n]; } field; (where n = 4 to 32740 and n is an even number)
O - DBCS open variable length VARLEN keyword	4-32740	none	_Packed struct { short len; char data[n]; } field; (where n = 4 to 32740)
T - time	8	none	char field[8];
L - date	6, 8, or 10	none	char field[n]; (where n = 6, 8 or 10)

Table 7. DDS-to-C/C++ Data Type Mappings (continued)			
DDS Data Type	**Length**	**Decimal Position**	**C/C++ Declaration**
Z - time stamp	26	none	char field[26];
[1]The C declaration (float or double) is based on what is specified in the FLTPCN (floating-point precision) keyword in the DDS: *SINGLE (default) is float, *DOUBLE is double.			

You can find more information in the *DDS Reference* at the IBM i Information Center web site:

```
http://www.ibm.com/systems/i/infocenter
```

Control Characters

Read this section for details on internal hexadecimal representation for control characters.

The following table identifies the internal hexadecimal representation of operating system control sequences used by the ILE C/C++ compiler and library.

Table 8. Internal Hexadecimal Representation

Print representation	**Internal representation**
NUL (null)	0x00
SOH (start of heading)	0x01
STX (start of text)	0x02
ETX (end of text)	0x03
SEL (select)	0x04
HT (horizontal tab)	0x05
RNL (required new line)	0x06
DEL (delete)	0x07
GE (graphic escape)	0x08
SPS (superscript)	0x09
RPT (repeat)	0x0a
VT (vertical tab)	0x0b
FF (form feed)	0x0c
CR (carriage return)	0x0d
SO (shift out)	0x0e
SI (shift in)	0x0f
DLE (data link escape)	0x10
DC1 (device control 1)	0x11
DC2 (device control 2)	0x12
DC3 (device control 3)	0x13
RES/ENP (restore or enable presentation)	0x14
NL (new line)	0x15
BS (backspace)	0x16
POC (program-operator communication)	0x17
CAN (cancel)	0x18
EM (end of medium)	0x19
UBS (unit backspace)	0x1a
CU1 (customer use 1)	0x1b
IFS (interchange file separator)	0x1c

Table 8. Internal Hexadecimal Representation (continued)	
Print representation	**Internal representation**
IGS (interchange group separator)	0x1d
IRS (interchange record separator)	0x1e
IUS/ITB (interchange unit separator or intermediate transmission block)	0x1f
DS (digit select)	0x20
SOS (start of significance)	0x21
FS (field separator)	0x22
WUS (word underscore)	0x23
BYP/INP (bypass or inhibit presentation)	0x24
LF (line feed)	0x25
ETB (end of transmission block)	0x26
ESC (escape)	0x27
SA (set attributes)	0x28
SM/SW (set mode or switch)	0x2a
CSP (control sequence prefix)	0x2b
MFA (modify field attribute)	0x2c
ENQ (enquiry)	0x2d
ACK (acknowledge)	0x2e
BEL (bell)	0x2f
SYN (synchronous idle)	0x32
IR (index return)	0x33
PP (presentation position)	0x34
TRN	0x35
NBS (numeric backspace)	0x36
EOT (end of transmission)	0x37
SBS (subscript)	0x38
IT (indent tab)	0x39
RFF (required form feed)	0x3a
CU3 (customer use 3)	0x3b
DC4 (device control 4)	0x3c
NAK (negative acknowledge)	0x3d
SUB (substitute)	0x3f
(blank character)	0x40

Related information

Read this section for information about related topics.

For additional information about topics related to ILE C/C++ programming, refer to the following IBM publications:

- CL Programming, SC41-5721-06 section in the *Programming* category at the IBM i Information Web site provides a wide-ranging discussion of IBM i programming topics. Topics include a general discussion on objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.
- GDDM Programming Guide, SC41-0536-00, provides information about using IBM i Graphical Data Display Manager (GDDM) to write graphics application programs. Includes many example programs and information to help users understand how the product fits into data processing systems.
- GDDM Reference, SC41-3718-00, provides information about using IBM i Graphical Data Display Manager (GDDM) to write graphics application programs. This manual provides detailed descriptions of all graphics functions available in GDDM. Also provides information about high-level language interfaces to GDDM.
- IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide, SC09-2712-07, provides programming information about the ILE C/C++ compiler. It includes programming considerations for interlanguage program and procedure calls, locales, handling exceptions, database, and device files. Examples are provided and performance tips for programming are also discussed.
- IBM Rational Development Studio for i: ILE C/C++ Language Reference, SC09-7852-03, provides reference information about the ILE C/C++ compiler, including elements of the language, statements, and preprocessor directives. Examples are provided and considerations for programming are also discussed.
- ILE C/C++ Runtime Library Functions, SC41-5607-05, provides reference information about ILE C/C++ library functions, including Standard C library functions and ILE C/C++ library extensions. Examples are provided and considerations for programming are also discussed.
- ILE Concepts, SC41-5606-08, explains concepts and terminology pertaining to the Integrated Language Environment architecture of the IBM i. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- The Application programming interfaces section in the *Programming* category at the IBM i Information Web site, provides information for the experienced application and system programmers who want to use the application programming interfaces (APIs). Provides getting started information and examples to help the programmer use APIs.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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 Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
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 is for planning purposes only. The information herein is subject to change before the products described become available.

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 illustrate 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. _enter the year or years_.

Programming interface information

This ILE C/C++ Compiler Reference publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

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 www.ibm.com/legal/copytrade.shtml.

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

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Oracle, Inc. in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Index

Special Characters

A

C

D

E

F

H

I

L

M

N

O

P

Q

S

T

U

W

Product Number: 5770-WDS

SC09-4816-08