IBM

eserver®

iSeries
WebSphere® Development Studio

ILE C/C++ Compiler Reference

Version 5

SC09-4816-03

IBM

eserver®

iSeries
WebSphere® Development Studio

ILE C/C++ Compiler Reference

Version 5

SC09-4816-03

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 165.

Fourth Edition (April 2004)

This edition applies to Version 5, Release 3, Modification Level 0, of IBM WebSphere Development Studio for iSeries (program 5722–WDS), ILE C/C++ compilers, and to all subsequent releases and modifications until otherwise indicated in new editions.

Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. You can send them to: compinfo@ca.ibm.com

Include the title and order number of this book, and the page number or topic related to your comment. Please be sure to include your e-mail address if you want a reply.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About This Book

This book contains reference information on:

- Using preprocessor statements in your program.
- Macros defined by the ILE C/C++ compiler.
- Pragmas recognized by the ILE C/C++ compiler.
- Command lines options for both native iSeries and Qshell working environments.
- I/O considerations for the iSeries environment.

Who Should Use This Guide

This guide 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. You need experience in using applicable iSeries™ menus and displays or Control Language (CL) commands. You also need knowledge of ILE as explained in the *ILE Concepts* manual.

Prerequisite and Related Information

Use the iSeries Information Center as your starting point for looking up iSeries and AS/400 Advanced Series technical information. You can access the Information Center from the following Web site:

```
http://www.ibm.com/eserver/iseries/infocenter
```

The iSeries Information Center contains advisors and important topics such as CL commands, system application programming interfaces (APIs), logical partitions, clustering, Java™ , TCP/IP, Web serving, and secured networks. It also includes links to related IBM® Redbooks and Internet links to other IBM Web sites such as the Technical Studio and the IBM home page.

Other information is listed in the “Bibliography” on page 163.

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 iSeries 400® Information Center Web site:

`http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm`

You need object authority to use CL commands. For more information on object authority, see the *Basic System Security and Planning* section in the *System Administration, Availability, and Maintenance* 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).

```
►►──statement──required_item──────────────────────────────────────────►◄
```

- Optional items appear below the main path.

```
►►──statement──┬───────────────┬──────────────────────────────────────►◄
               └─optional_item─┘
```

- 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.

```
►►──statement──┬─required_choice1─┬───────────────────────────────────►◄
               └─required_choice2─┘
```

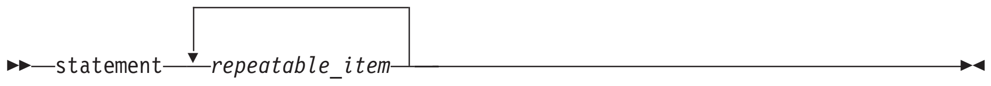
  If choosing one of the items is optional, the entire stack appears below the main path.

```
►►──statement──┬──────────────────┬───────────────────────────────────►◄
               ├─optional_choice1─┤
               └─optional_choice2─┘
```

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

```
               ┌─default_item───┐
►►──statement──┴─alternate_item─┴─────────────────────────────────────►◄
```

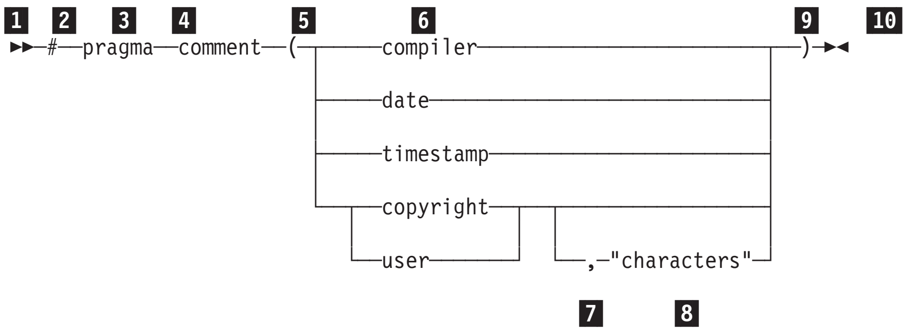
- An arrow returning to the left above the main line indicates an item that can be repeated.



  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 "Pragma Directives (#pragma)" on page 16 for information on 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")
```

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and high-quality information. IBM welcomes any comments about this book or any other iSeries documentation.

- If you prefer to send comments by mail, use the following address:

  IBM Canada Ltd. Laboratory
  Information Development
  2G/KB7/1150/TOR
  1150 Eglinton Avenue East
  Toronto, Ontario, Canada M3C 1H7

  If you are mailing a readers' comment form from a country other than the United States, you can give the form to the local IBM branch office or IBM representative for postage-paid mailing.
- If you prefer to send comments electronically, use one of these e-mail addresses:
  - Comments on books:

    torrcf@ca.ibm.com
  - Comments on the iSeries 400 Information Center:

    RCHINFOC@us.ibm.com

Be sure to include the following:

- The name of the book
- The publication number of the book
- The page number or topic to which your comment applies.

Chapter 1. Preprocessor Directives

This chapter describes the C/C++ preprocessor directives.

Preprocessor Overview

Preprocessing is a preliminary operation on C and C++ files before they are passed to the compiler. Preprocessing lets you:

- Replace tokens in the current file with specified replacement tokens
- Imbed files within the current file
- Conditionally compile sections of the current file
- Generate diagnostic messages
- Change the line number of the next line of source and change the file name of the current file
- Apply machine-specific rules to specified sections of code

A *token* is a series of characters delimited by white space. The only white space allowed on a preprocessor directive is the space, horizontal tab, vertical tab, form feed, and comments. The new-line character can also separate preprocessor tokens.

The preprocessed source program file must be a valid C or C++ program.

The preprocessor is controlled by the following directives:

#define	Defines a preprocessor macro.
#undef	Removes a preprocessor macro definition.
#error	Defines text for a compile-time error message.
#include	Inserts text from another source file.
#if	Conditionally suppresses portions of source code, depending on the result of a constant expression.
#ifdef	Conditionally includes source text if a macro name is defined.
#ifndef	Conditionally includes source text if a macro name is not defined.
#else	Conditionally includes source text if the previous **#if**, **#ifdef**, **#ifndef**, or **#elif** test fails.
#elif	Conditionally includes source text if the previous **#if**, **#ifdef**, **#ifndef**, or **#elif** test fails, depending on the value of a constant expression.
#endif	Ends conditional text.
#line	Supplies a line number for compiler messages.
#pragma	Specifies implementation-defined instructions to the compiler.

Preprocessor Directive Format

Preprocessor directives begin with the # token followed by a preprocessor keyword. The # token must appear as the first character that is not white space on a line. The # is not part of the directive name and can be separated from the name with white spaces.

A preprocessor directive ends at the new-line character unless the last character of the line is the \ (backslash) character. If the \ character appears as the last character in the preprocessor line, the preprocessor interprets the \ and the new-line character as a continuation marker. The preprocessor deletes the \ (and the following new-line character) and splices the physical source lines into continuous logical lines.

Except for some **#pragma** directives, preprocessor directives can appear anywhere in a program.

Macro Directives and Operators (#define, #undef, #, ##)

A macro is a literal name that can be assigned a value. Before a program is compiled, the preprocessor substitutes occurences of each macro in program source code with that macro's assigned value.

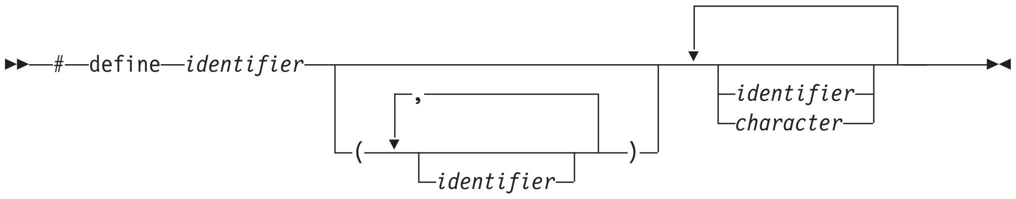
Macros can be predefined by the operating system or the compiler. See Chapter 2, "Predefined Macros," on page 17 for more information on these.

Macros can also be defined in program source code, as described below.

#define (Defining and Expanding a Macro)

The preprocessor **define** directive directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.

The **define** directive has the form:



The **define** directive can contain an object-like definition or a function-like definition.

Object-Like Macros

An *object-like macro definition* replaces a single identifier with the specified replacement tokens. The following object-like definition causes the preprocessor to replace all subsequent instances of the identifier COUNT with the constant 1000:

```
#define COUNT 1000
```

If the statement

```
int arry[COUNT];
```

appears after this definition and in the same file as the definition, the preprocessor would change the statement to

```
int arry[1000];
```

in the output of the preprocessor.

Other definitions can make reference to the identifier `COUNT`:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of `MAX_COUNT` with `COUNT + 100`, which the preprocessor then replaces with `1000 + 100`.

If a number that is partially built by a macro expansion is produced, the preprocessor does not consider the result to be a single value. For example, the following will not result in the value 10.2 but in a syntax error.

```
#define a 10
a.2
```

Using the following also results in a syntax error:

```
#define a 10
#define b a.11
```

Identifiers that are partially built from a macro expansion may not be produced. Therefore, the following example contains two identifiers and results in a syntax error:

```
#define d efg
abcd
```

Function-Like Macros

A *function-like macro definition* replaces a single identifier with the result of a specified function.

Defining:
: An identifier is followed by a parenthesized parameter list and the replacement tokens. The parameters are imbedded in the replacement code. White space cannot separate the identifier (which is the name of the macro) and the left parenthesis of the parameter list. A comma must separate each parameter. For portability, you should not have more than 31 parameters for a macro.

Invoking:
: An identifier is followed by a list of arguments in parentheses. A comma must separate each argument. Once the preprocessor identifies a function-like macro invocation, argument substitution takes place. A parameter in the replacement code is replaced by the corresponding argument. Any macro invocations contained in the argument itself are completely replaced before the argument replaces its corresponding parameter in the replacement code.

The following line defines the macro `SUM` as having two parameters `a` and `b` and the replacement tokens `(a + b)`:

```
#define SUM(a,b) (a + b)
```

This definition would cause the preprocessor to change the following statements (if the statements appear after the previous definition):

```
c = SUM(x,y);
c = d * SUM(x,y);
```

In the output of the preprocessor, these statements would appear as:

```
c = (x + y);
c = d * (x + y);
```

Use parentheses to ensure correct evaluation of replacement text. For example, the definition:

```
#define SQR(c)  ((c) * (c))
```

requires parentheses around each parameter `c` in the definition in order to correctly evaluate an expression like:

```
y = SQR(a + b);
```

The preprocessor expands this statement to:

```
y = ((a + b) * (a + b));
```

Without parentheses in the definition, the correct order of evaluation is not preserved, and the preprocessor output is:

```
y = (a + b * a + b);
```

Arguments of the `#` and `##` operators are converted *before* replacement of parameters in a function-like macro.

The number of arguments in a macro invocation must be the same as the number of parameters in the corresponding macro definition.

Commas in the macro invocation argument list do not act as argument separators when they are:
- in character constants
- in string literals
- surrounded by parentheses

Once defined, a preprocessor identifier remains defined and in scope independent of the scoping rules of the language. The scope of a macro definition begins at the definition and does not end until a corresponding **undef** directive is encountered. If there is no corresponding **undef** directive, the scope of the macro definition lasts until the end of the compilation unit.

A recursive macro is not fully expanded. For example, the definition

```
   #define x(a,b) x(a+1,b+1) + 4
```

expands

```
   x(20,10)
```

to

```
   x(20+1,10+1) + 4
```

rather than trying to expand the macro `x` over and over within itself. After the macro `x` is expanded, it is a call to function `x()`.

A definition is not required to specify replacement tokens. The following definition removes all instances of the token `debug` from subsequent lines in the current file:

```
#define debug
```

You can change the definition of a defined identifier or macro with a second preprocessor **define** directive only if the second preprocessor **define** directive is preceded by a preprocessor **undef** directive. The **undef** directive nullifies the first definition so that the same identifier can be used in a redefinition.

Within the text of the program, the preprocessor does not scan character constants or string constants for macro invocations.

Example: #define directive

The following program contains two macro definitions and a macro invocation that refers to both of the defined macros:

```
/**
 ** This example illustrates #define directives.
 **/


#include <stdio.h>

#define SQR(s)  ((s) * (s))
#define PRNT(a,b) \
  printf("value 1 = %d\n", a); \
  printf("value 2 = %d\n", b) ;

int main(void)
{
  int x = 2;
  int y = 3;

     PRNT(SQR(x),y);

  return(0);
}
```

After being interpreted by the preprocessor, this program is replaced by code equivalent to the following:

```
#include <stdio.h>

int main(void)
{
  int x = 2;
  int y = 3;

     printf("value 1 = %d\n", ( (x) * (x) ) );
     printf("value 2 = %d\n", y);

  return(0);
}
```

The program produces the following output:

```
value 1 = 4
value 2 = 3
```

#undef (Undefining a Macro)

The *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition.

The **undef** directive has the form:

```
►►──#──undef──identifier───────────────────────────────────────────►◄
```

If the identifier is not currently defined as a macro, **undef** is ignored.

Example: #undef directive

The following directives define BUFFER and SQR:

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

The following directives nullify these definitions:

```
#undef BUFFER
#undef SQR
```

Any occurrences of the identifiers BUFFER and SQR that follow these **undef** directives are not replaced with any replacement tokens. Once the definition of a macro has been removed by an **undef** directive, the identifier can be used in a new **define** directive.

# Operator

The # (single number sign) operator converts a parameter of a function-like macro into a character string literal.

For example, if macro ABC is defined using the following directive:

```
   #define ABC(x)   #x
```

all subsequent invocations of the macro ABC would be expanded into a character string literal containing the argument passed to ABC. For example:

Invocation	Result of Macro Expansion
ABC(1)	"1"
ABC(Hello there)	"Hello there"

The # operator should not be confused with the null directive.

Use the # operator in a function-like macro definition according to the following rules:

- A parameter following # operator in a function-like macro is converted into a character string literal containing the argument passed to the macro.
- White-space characters that appear before or after the argument passed to the macro are deleted.
- Multiple white-space characters imbedded within the argument passed to the macro are replaced by a single space character.
- If the argument passed to the macro contains a string literal and if a \ (backslash) character appears within the literal, a second \ character is inserted before the original \ when the macro is expanded.
- If the argument passed to the macro contains a " (double quotation mark) character, a \ character is inserted before the " when the macro is expanded.
- The conversion of an argument into a string literal occurs before macro expansion on that argument.

- If more than one ## operator or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.
- If the result of the macro expansion is not a valid character string literal, the behavior is undefined.

Example: # operator

The following examples demonstrate the use of the # operator:

```
#define STR(x)        #x
#define XSTR(x)       STR(x)
#define ONE           1
```

Invocation	Result of Macro Expansion
`STR(\n "\n" '\n')`	`"\n \"\\n\" '\\n'"`
`STR(ONE)`	"ONE"
`XSTR(ONE)`	"1"
`XSTR("hello")`	"\"hello\""

## Operator (Macro Concatenation)

The ## (double number sign) operator concatenates two tokens in a macro invocation (text and/or arguments) given in a macro definition.

If a macro XY was defined using the following directive:

```
#define XY(x,y)    x##y
```

the last token of the argument for x is concatenated with the first token of the argument for y.

For example,

Invocation	Result of Macro Expansion
`XY(1, 2)`	`12`
`XY(Green, house)`	`Greenhouse`

Use the ## operator according to the following rules:

- The ## operator cannot be the very first or very last item in the replacement list of a macro definition.
- The last token of the item in front of the ## operator is concatenated with first token of the item following the ## operator.
- Concatenation takes place before any macros in arguments are expanded.
- If the result of a concatenation is a valid macro name, it is available for further replacement even if it appears in a context in which it would not normally be available.
- If more than one ## operator and/or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.

Example: ## operator

The following examples demonstrate the use of the ## operator:

```
#define ArgArg(x, y)          x##y
#define ArgText(x)            x##TEXT
#define TextArg(x)            TEXT##x
```

```
#define TextText             TEXT##text
#define Jitter               1
#define bug                  2
#define Jitterbug            3
```

Invocation	Result of Macro Expansion
`ArgArg(lady, bug)`	`ladybug`
`ArgText(con)`	`conTEXT`
`TextArg(book)`	`TEXTbook`
`TextText`	`TEXTtext`
`ArgArg(Jitter, bug)`	`3`

Preprocessor Error Directive (#error)

A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail.

The **error** directive has the form:

```
              ┌────────────┐
              ▼            │
►►──#──error────character──┴──────────────────────────────────────────►◄
```

Use the **error** directive as a safety check during compilation. For example, if your program uses preprocessor conditional compilation directives, put **error** directives in the source file to prevent code generation if a section of the program is reached that should be bypassed.

For example, the directive

```
#error Error in TESTPGM1 - This section should not be compiled
```

generates the following error message:

```
Error in TESTPGM1 - This section should not be compiled
```

File Inclusion (#include)

A *preprocessor include directive* causes the preprocessor to replace the directive with the contents of the specified file. The **include** directive has the form:

```
►►──#include──┬─<filename>─┬──────────────────────────────────────────►◄
              └─"filename"─┘
```

The following table indicates the search path compiler takes for source physical files. See the default file names and search paths below.

Filename	Member	File	Library
mbr	*mbr*	default file	default search
file/mbr[1]	*mbr*	*file*	default search
mbr.file	*mbr*	*file*	default search
lib/file/mbr	*mbr*	*file*	*lib*
lib/file(mbr)	*mbr*	*file*	*lib*

Filename	Member	File	Library
Note: [1] If the include file format <file/mbr.h> is used, the compiler searches for *mbr* in the file in the library list first. If *mbr* is not found, then the compiler searches for *mbr.h* in the same file in the library list. Only ″h″ or ″H″ are allowed as member name extensions.			

If library and file are not specified, the preprocessor uses a specific search path depending on which delimiter surrounds the *filename*. The < > delimiter specifies the name as a system include file. The " " delimiter specifies the name as a user include file.

The following describes the search paths for the #include directive used by the compiler.

- Default File Names When the Library and File are not Named (member name only):

Include Type	**Default File Name**
< >	QCSRC
" "	The source file of the root source member, where root source member is the library, file, and member determined by the SRCFILE option of the Create Module or Create Bound Program commands.

- Default Search Paths When the Filename is not Library Qualified:

Include Type	**Search Path**
< >	Searches the current library list (*LIBL)
" "	Checks the library containing the root source member; if not found there, the compiler searches the user portion of the library list, using either the filename specified or the file name of the root source member (if no filename is specified); if not found, the compiler searches the library list (*LIBL) using the specified filename.

- Search Paths When the Filename is Library Qualified (lib/file/mbr):

Include Type	**Search Path**
< >	Searches for lib/file/mbr only
" "	Searches for the member in the library and file named. If not found, searches the user portion of the library list, using the file and member names specified.

User includes are treated the same as system includes when the *SYSINCPATH option has been specified with the Create Module or Create Bound Program commands.

The preprocessor resolves macros on a `#include` directive. After macro replacement, the resulting token sequence must consist of a file name enclosed in either double quotation marks or the characters < and >. For example:

```
#define MONTH <july.h>
#include MONTH
```

Usage

If there are a number of declarations used by several files, you can place all these definitions in one file and `#include` that file in each file that uses the definitions. For example, the following file `defs.h` contains several definitions and an inclusion of an additional file of declarations:

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "mydefs.h"
```

You can embed the definitions that appear in `defs.h` with the following directive:

```
#include "defs.h"
```

One of the ways you can combine the use of preprocessor directives is demonstrated in the following example. A `#define` is used to define a macro that represents the name of the C or C++ standard I/O header file. A `#include` is then used to make the header file available to the C or C++ program.

```
#define  IO_HEADER   <stdio.h>
      .
      .
      .
#include IO_HEADER   /* equivalent to specifying #include <stdio.h> */
      .
      .
      .
```

Using the #include Directive When Compiling Source in an Integrated File System File

You can use the `SRCSTMF` keyword to specify an Integrated File System file at compile time. The #include processing differs from source physical file processing in that the library list is not searched. The search path specified by the INCLUDE environment variable (if it is defined), and the compiler's default search path are used to resolve header files.

The compiler's default include path is `/QIBM/include`.

#include files use the delimiters "" or <>.

When attempting to open the include file, the compiler searches in turn each directory in the search path until the file is found or all search directories have been exhausted.

The algorithm to search for include files is:

```
if file is fully qualified (a slash / starts the name) then
  attempt to open the fully qualified file
else
  if "" is delimiter, check job's current directory
  if not found:
    loop through the list of directories specified in the INCLUDE
          environment variable and then the default include path
    until the file is found or the end of the include path is encountered
endif
```

For more information, refer to *Using the ILE C/C++ Stream Functions With the iSeries Integrated File System* in *WebSphere Development Studio: ILE C/C++ Programmer's Guide*.

Conditional Compilation Directives

A *preprocessor conditional compilation directive* causes the preprocessor to conditionally suppress the compilation of portions of source code. These directives test a constant expression or an identifier to determine which tokens the preprocessor should pass on to the compiler and which tokens should be bypassed during preprocessing. The directives are:

- **#if**
- **#ifdef**
- **#ifndef**
- **#elif**
- **#else**
- **#endif**

The preprocessor conditional compilation directive spans several lines:

- The condition specification line
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional)
- The **#elif** line (optional)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional)
- The **#else** line (optional)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to zero (optional)
- The preprocessor **#endif** directive

For each **if**, **ifdef**, and **ifndef** directive, there are zero or more **elif** directives, zero or one **else** directive, and one matching **endif** directive. All the matching directives are considered to be at the same nesting level.

You can nest conditional compilation directives. In the following directives, the first **#else** is matched with the **#if**.

```
#ifdef MACNAME
                 /*  tokens added if MACNAME is defined              */
#   if TEST <=10
                 /* tokens added if MACNAME is defined and TEST <= 10 */
#   else
                 /* tokens added if MACNAME is defined and TEST >  10 */
```

```
#   endif
#else
                /*  tokens added if MACNAME is not defined          */
#endif
```

Each directive controls the block immediately following it. A block consists of all the tokens starting on the line following the directive and ending at the next conditional compilation directive at the same nesting level.

Each directive is processed in the order in which it is encountered. If an expression evaluates to zero, the block following the directive is ignored.

When a block following a preprocessor directive is to be ignored, the tokens are examined only to identify preprocessor directives within that block so that the conditional nesting level can be determined. All tokens other than the name of the directive are ignored.

Only the first block whose expression is nonzero is processed. The remaining blocks at that nesting level are ignored. If none of the blocks at that nesting level has been processed and there is an **else** directive, the block following the **else** directive is processed. If none of the blocks at that nesting level has been processed and there is no **else** directive, the entire nesting level is ignored.

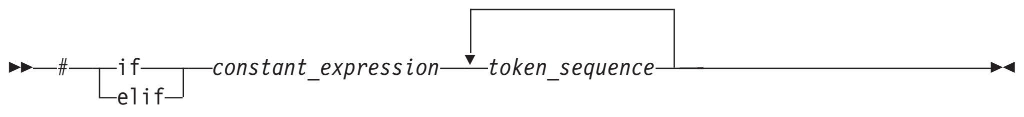
#if, #elif

The **if** and **elif** directives compare the value of the expression to zero.

If the constant expression evaluates to a nonzero value, the tokens that immediately follow the condition are passed on to the compiler.

If the expression evaluates to zero and the conditional compilation directive contains a preprocessor **elif** directive, the source text located between the **elif** and the next **elif** or **else** directive is selected by the preprocessor to be passed on to the compiler. The **elif** directive cannot appear after the **else** directive.

All macros are expanded, any `defined()` expressions are processed and all remaining identifiers are replaced with the token `0`.



The expressions that are tested must be integer constant expressions with the following properties:

- No casts are performed.
- Arithmetic is performed using **long int** values.
- The expression can contain defined macros. No other identifiers can appear in the expression.
- The constant expression can contain the unary operator **defined**. This operator can be used only with the preprocessor keyword **if** or **elif**. The following expressions evaluate to `1` if the *identifier* is defined in the preprocessor, otherwise to `0`:

  ```
  defined identifier
  defined(identifier)
  ```

For example:

```
#if defined(TEST1) || defined(TEST2)
```

Note: If a macro is not defined, a value of 0 (zero) is assigned to it. In the following example, TEST must be a macro identifier:

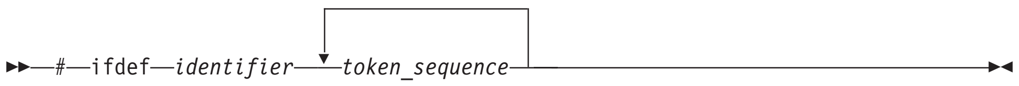
```
#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n", array[i]);
#elif TEST < 0
   printf("array subscript out of bounds \n");
#endif
```

#ifdef

The **ifdef** directive checks for the existence of macro definitions.

If the identifier specified is defined as a macro, the tokens that immediately follow the condition are passed on to the compiler after a newline.

The **ifdef** directive has the form:



The following example defines MAX_LEN to be 75 if EXTENDED is defined for the preprocessor. Otherwise, MAX_LEN is defined to be 50.

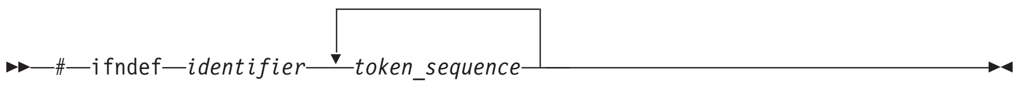
```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

#ifndef

The **ifndef** directive checks for the existence of macro definitions.

If the identifier specified is not defined as a macro, the tokens that immediately follow the condition are passed on to the compiler after a newline.

The **ifndef** directive has the form:



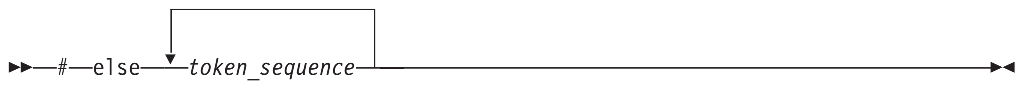
An identifier must follow the **#ifndef** keyword. The following example defines MAX_LEN to be 50 if EXTENDED is not defined for the preprocessor. Otherwise, MAX_LEN is defined to be 75.

```
#ifndef EXTENDED
#   define MAX_LEN 50
#else
#   define MAX_LEN 75
#endif
```

#else

If the condition specified in the **if**, **ifdef**, or **ifndef** directive evaluates to `0`, and the conditional compilation directive contains an **else** directive, the source text located between the **else** and the **endif** directives is selected by the preprocessor to be passed on to the compiler.

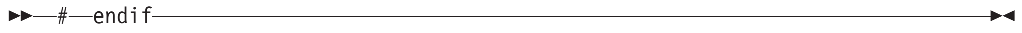
The **else** directive has the form:



#endif

The **endif** directive ends the conditional compilation directive.

It has the form:



Examples: Conditional compilation directives

The following example shows how you can nest preprocessor conditional compilation directives:

```
#if defined(TARGET1)
#   define SIZEOF_INT 16
#   ifdef PHASE2
#      define MAX_PHASE 2
#   else
#      define MAX_PHASE 8
#   endif
#elif defined(TARGET2)
#   define SIZEOF_INT 32
#   define MAX_PHASE 16
#else
#   define SIZEOF_INT 32
#   define MAX_PHASE 32
#endif
```

The following program contains preprocessor conditional compilation directives:

```
/**
 ** This example contains preprocessor
 ** conditional compilation directives.
 **/

#include <stdio.h>

int main(void)
{
   static int array[ ] = { 1, 2, 3, 4, 5 };
   int i;

   for (i = 0; i <= 4; i++)
   {
      array[i] *= 2;

#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n", array[i]);
#endif
```

```
   }
   return(0);
}
```

Line Control (#line)

A *preprocessor line control directive* supplies line numbers for compiler messages. It causes the compiler to view the line number of the next source line as the specified number.

The **line** directive has the form:

```
►►──#──┬──────┬──┬─decimal_constant──┬───────────────────┬─┬────────────────────►◄
       └─line─┘  │                   └─"──file_name──"───┘ │
                 └─characters──────────────────────────────┘
```

In order for the compiler to produce meaningful references to line numbers in preprocessed source, the preprocessor inserts **line** directives where necessary (for example, at the beginning and after the end of included text).

A file name specification enclosed in double quotation marks can follow the line number. If you specify a file name, the compiler views the next line as part of the specified file. If you do not specify a file name, the compiler views the next line as part of the current source file.

The token sequence on a **line** directive is subject to macro replacement. After macro replacement, the resulting character sequence must consist of a decimal constant, optionally followed by a file name enclosed in double quotation marks.

Example: line directive

You can use **#line** control directives to make the compiler provide more meaningful error messages. The following program uses **#line** control directives to give each function an easily recognizable line number:

```
/**
 ** This example illustrates #line directives.
 **/

#include <stdio.h>
#define LINE200 200

int main(void)
{
   func_1();
   func_2();
}

#line 100
func_1()
{
   printf("Func_1 - the current line number is %d\n",_ _LINE_ _);
}

#line LINE200
```

```
func_2()
{
   printf("Func_2 - the current line number is %d\n",_ _LINE_ _);
}
```

This program produces the following output:

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

Null Directive (#)

The *null directive* performs no action. It consists of a single # on a line of its own.

The null directive should not be confused with the # operator or the character that starts a preprocessor directive.

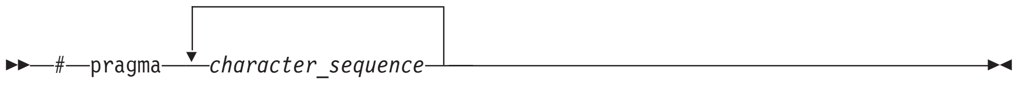
Example: # (null) directive

If MINVAL is a defined macro name, no action is performed. If MINVAL is not a defined identifier, it is defined 1.

```
#ifdef MINVAL
  #
#else
  #define MINVAL 1
#endif
```

Pragma Directives (#pragma)

A *pragma* is an implementation-defined instruction to the compiler. It has the general form:



where *character_sequence* is a series of characters giving a specific compiler instruction and arguments, if any.

Unless specifically noted otherwise, character sequences in pragma directives are not case sensitive. For example, the following two pragma directives are functionally equivalent:

```
#pragma convert(37)
#pragma CoNvErT(37)
```

The *character_sequence* on a pragma is subject to macro substitutions. For example,

```
#define XX_ISO_DATA isolated_call(LG_ISO_DATA)
// ...
#pragma XX_ISO_DATA
```

More than one pragma construct can be specified on a single **pragma** directive. The compiler ignores unrecognized pragmas.

ILE C/C++ pragmas are described in Chapter 3, “ILE C/C++ Pragmas,” on page 21.

Chapter 2. Predefined Macros

The ILE C/C++ compiler recognizes the predefined macros described in this chapter.

- "ANSI/ISO Standard Predefined Macros"
- "ILE C/C++ Predefined Macros" on page 18

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 will be 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` will be 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__`
: Defined if the C compiler conforms to the ANSI standard. This macro is undefined if the language level is set to anything other than `ANSI`.

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

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

where:
- `hh` represents the hour.
- `mm` represents the minutes.
- `ss` represents the seconds.

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

Notes:

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__ will change during compilation as the compiler processes subsequent lines of your source program.
4. The value of __FILE__, and __TIME__ will change 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 will display the values of the predefined macros __LINE__, __FILE__, __TIME__, and __DATE__ and will print a message indicating the program's conformance 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

- "#define (Defining and Expanding a Macro)" on page 2
- "#undef (Undefining a Macro)" on page 5
- "Line Control (#line)" on page 15

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 for the Create Module and Create Bound Program commands 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) is in effect.
_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	Defined when the **#pragma chars(unsigned)** directive is in effect, or when the DFTCHAR compiler option is set to *UNSIGNED. Indicates default character type is unsigned.

`__cplusplus98__interface__`	Defined by C++ compiler when the LANGLVL(*ANSI) compiler option is specified.
`__EXTENDED__`	Defined when the LANGLVL(*ANSI) compiler option is not 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 iSeries processor.
`__HOS_OS400__`	C++ Indicates that the host operating system is OS/400.
`__IBMCPP__`	C++ Indicates the version number of the ILE C/C++ compiler.
`__IFS_IO__`	Defined when SYSIFCOPT(*IFSIO) or SYSIFCOPT(*IFS64IO) is specified on the Create Module or Create Bound Program commands.
`__IFS64_IO__`	Defined when SYSIFCOPT(*IFS64IO) is specified on the Create Module or Create Bound Program commands. When this macro is defined, _LARGE_FILES and _LARGE_FILE_API are also defined in the relevant IBM-supplied header files.
`__ILEC400__`	C Defined only by the compiler. You can use this macro in source code that is compiled for several platforms. Mark code that is to be compiled only for the iSeries platform with `#ifdef __ILEC400__` or, `#if defined(__ILEC400__)` preprocessor directives.
`__ILEC400_TGTVRM__`	C Defined by the compiler as an integral value that maps to the version/release/modification of the OS/400® that the module or program being compiled is intended to run on. The target release, VxRyMz, translates to an __ILEC400_TGTVRM__ value of xyz, where x, y, and z are integer values. For example, a target release of V3R7M0 will cause the macro to have an integral value of 370.
`_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__`	C++ Defined when the RTBND(*LLP64) compiler option is in effect.
`__OS400__`	This macro is always defined when the compiler is used with the OS/400 operating system.
`__OS400_TGTVRM__`	Defined only by the compiler as an integral value that maps to the version/release/modification of the OS/400® that the module or program being compiled is intended to run on. The target release, VxRyMz, translates to an __OS400_TGTVRM__ value of xyz, where x, y, and z are integer values.

`__POSIX_LOCALE__`	Defined when the LOCALETYPE(*LOCALE) or LOCALETYPE(*LOCALEUCS2) compiler options are specified.
`__RTTI_DYNAMIC_CAST__`	Defined when the OPTION(*RTTIALL) or OPTION(*RTTICAST) compiler options are specified, for C++ programs only. This macro is not defined for C.
`__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 iSeries processor.
`__TIMESTAMP__`	A character string literal containing the date and time when the source file was last changed. The date and time will be 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` will be a blank character. `hh` represents the hour. `mm` represents the minutes. `ss` represents the seconds. `yyyy` represents the year. **Note:** Other compilers may not supported this macro. If the macro is supported on other compilers, the date and time values may be different than those that are shown here.
`__TOS_OS400__`	Indicates that the target operating system is OS/400.
`__UCS2__`	Defined when LOCALETYPE(*LOCALEUCS2) is specified on the Create Module or Create Bound Program commands.
`__UTF32__`	Defined when LOCALETYPE(*LOCALEUTF) is specified on the Create Module or Create Bound Program commands.
`__wchar_t`	C Defined by the standard header file stddef.h. C++ Defined by the C++ compiler.

Chapter 3. ILE C/C++ Pragmas

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

Pragma Name	Valid with C	Valid with C++
"argopt" on page 23	✓	✓
"argument" on page 25	✓	
"cancel_handler" on page 27	✓	✓
"chars" on page 28	✓	✓
"checkout" on page 29	✓	
"comment" on page 30	✓	✓
"convert" on page 31	✓	
"datamodel" on page 32	✓	✓
"define" on page 34		✓
"descriptor" on page 35	✓	✓
"disable_handler" on page 37	✓	✓
"disjoint" on page 38		✓
"enum" on page 39	✓	✓
"exception_handler" on page 43	✓	✓
"hashome" on page 46		✓
"implementation" on page 47		✓
"info" on page 48		✓
"inline" on page 49	✓	
"ishome" on page 50		✓
"isolated_call" on page 51		✓
"linkage" on page 52	✓	
"map" on page 54	✓	✓
"mapinc" on page 55	✓	
"margins" on page 58	✓	
"namemangling" on page 59		✓
"noargv0" on page 60	✓	
"noinline (function)" on page 61	✓	
"nomargins" on page 62	✓	
"nosequence" on page 63	✓	
"nosigtrunc" on page 64	✓	
"pack" on page 65	✓	✓
"page" on page 71	✓	
"pagesize" on page 72	✓	
"pointer" on page 73	✓	✓

Pragma Name	Valid with C	Valid with C++
"priority" on page 74		✓
"sequence" on page 75	✓	
"strings" on page 76	✓	✓
"weak" on page 77		✓

argopt

C C++

```
►►──#pragma argopt──(──┬─function_name──────────┬──)──────────────────────────►◄
                       ├─typedef_of_function_name─┤
                       ├─typedef_of_function_ptr──┤
                       └─function_ptr─────────────┘
```

Description

Argument Optimization (argopt) is a pragma which may improve run-time performance. Applied to a bound procedure, optimizations can be achieved by:

- Passing space pointer parameters in to general-purpose registers (GPRs).
- Storing a space pointer returned from a function in to 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 will be 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 is placed where the function, the function pointer, typedef of a function pointer or typedef of a function is visible (can be used) within a region of the program code called its scope. #pragma argopt scope is determined its placement in the code. An error will be issued when #pragma argopt is not within the scope of the declaration. The #pragma argopt directive can fall within file, block, or structure scope.

```
#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

```
►►─#─pragma─argument─(─function_name─┬─,─OS──────────────┬─)─────►◄
                                      │     └─,─nowiden─┘   │
                                      ├─,─VREF──────────────┤
                                      │       └─,─nowiden─┘ │
                                      └─,─nowiden───────────┘
```

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 on making calls to external programs, see pragma "linkage" on page 52.

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 usual *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 or _System keyword has already been specified for this function.
- The function has already been called prior to the #pragma argument directive.

cancel_handler

C C++

```
►►—#—pragma—cancel_handler—(—function_name—┬—,—0————————┬—)——————————►◄
                                           └—,—com_area—┘
```

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 *Run-Time Library Reference* 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.

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

chars

C C++

```
                       ┌─unsigned─┐
►►──#──pragma──chars──(──┴─signed───┴──)──────────────────────────────────►◄
```

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

```
►►──#──pragma──checkout──(──┬─suspend─┬──)─────────────────────────────────►◄
                            └─resume──┘
```

Description

Specifies whether or not the compiler should give compiler information when a CHECKOUT setting other than *NONE is specified for the Create Module or Create Bound Program commands.

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++

```
►►─#─pragma─comment─(─┬─┬─compiler──┬─────────────────────────────┬─)────────►◄
                      │ ├─date──────┤                             │
                      │ └─timestamp─┘                             │
                      └─┬─copyright─┬──┬───────────────────────┬──┘
                        └─user──────┘  └─,─"─characters─"──────┘
```

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++

```
▶▶─#─pragma─convert─(─ccsid─)──────────────────────────────────────▶◀
```

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 may range between 0 and 65535. See the *ILE C/C++ Run-Time Library Functions* manual for more information about code pages.

Notes on Usage

The run-time library functions that parse format strings (such as `printf()` and `scanf()`) cannot use ASCII format strings. Therefore, all format strings must be in EBCDIC.

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

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

If a CCSID with the value 65535 is specified, 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 is 65535 and the job is 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.

If the LOCALETYPE(*LOCALEUCS2) option is specified for the Create Module or Create Bound Program commands, wide-character literals are not converted. See *Using Unicode Support for Wide-Character Literals* in *WebSphere Development Studio: ILE C/C++ Programmer's Guide* for more information.

datamodel

C C++

```
                            ┌P128──┐
►►──#──pragma──datamodel──(─┼LLP64─┼─)──────────────────────────────►◄
                            └pop───┘
```

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	Default setting. If specified, sizes of int, long, and pointer types in program code following this pragma are 4, 4, and 16 bytes respectively, until another pragma datamodel setting is specified. If the RTBND(*LLP64) option is in effect, pointer types in code sections affected by this pragma will be 8 bytes in size.
LLP64, llp64	If specified, sizes of int, long, and pointer types in program code following this pragma are 4, 4, and 8 bytes respectively, until another pragma datamodel setting is specified.
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 *WebSphere Development Studio: ILE C/C++ Programmer's Guide*, and *Teraspace and single-level store* in *ILE Concepts*.

define

C++

```
►►──#──pragma──define──(──template_class_name──)──────────────────────────────►◄
```

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++

```
►►─#─pragma─descriptor─(─void─function_name─(─┤ od_specifiers ├─)─)────►◄
```

od_specifiers:

```
├──┬─""───┬──┬─────────────────────┬──────────────────────────────────────┤
   ├─void─┤  │ ┌─────────────────┐ │
   └─*────┘  │ ▼                 │ │
             └───,──┬─""───┬─────┴─┘
                    ├─void─┤
                    └─*────┘
```

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 iSeries 400 Information Center Web site:

```
http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm
```

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:
- 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:

- 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

▶ C ▶ C++

```
►►──#──pragma──disable_handler──────────────────────────────────────────►◄
```

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

C++

```
►►──#──pragma──disjoint──────────────────────────────────────────────►

                              ┌─────────────────────────┐
      ┌───┐                   ▼      ┌───┐              │
►─(───▼─*─┴───identifier──────┴──,───▼─*─┴───identifier─┴──)──────────►◄
```

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:
- 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.

enum

C C++

```
►►──#pragma──enum──(──┬─1──────────────┬──)─────────────────────────────────►◄
                      ├─2──────────────┤
                      ├─4──────────────┤
                      ├─int────────────┤
                      ├─small──────────┤
                      ├─pop────────────┤
                      ├─system_default─┤
                      └─user_default───┘
```

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 128.

Parameters

1, 2, 4 Specifies that enumerations be stored in 1, 2, or 4-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.

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.

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:

Range of Element Values	Enum Options				
	small (default)	**1**	**2**	**4**	**int**
0 .. 127	1 byte unsigned	1 byte signed	2 bytes signed	4 bytes signed	4 bytes signed
0 .. 255	1 byte unsigned	1 byte unsigned	2 bytes signed	4 bytes signed	4 bytes signed
-128 .. 127	1 byte signed	1 byte signed	2 bytes signed	4 bytes signed	4 bytes signed

0 .. 32767	2 bytes unsigned	ERROR	2 bytes signed	4 bytes signed	4 bytes signed
0 .. 65535	2 bytes unsigned	ERROR	2 bytes unsigned	4 bytes signed	4 bytes signed
-32768 .. 32767	2 bytes signed	ERROR	ERROR	4 bytes signed	4 bytes signed
0 .. 2147483647	4 bytes unsigned	ERROR	ERROR	4 bytes signed	4 bytes signed
0 .. 4294967295	4 bytes unsigned	ERROR	ERROR	4 bytes unsigned	C++ 4 bytes unsigned C ERROR
-2147483648 .. 2147483647	4 bytes signed	ERROR	4 bytes signed	4 bytes signed	4 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 example, the following code segments contain errors:

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

3. The enum constant range does not fit within the range of an unsigned int.

```
#pragma enum ( small )
   enum e_tag { a=0,
                b=4294967296 /* error: larger than maximum int */
              } e_var;
#pragma enum ( pop )
```

4. 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.

5. 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.
6. 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.
7. 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.
8. 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++

```
►►—#—pragma—exception_handler—(—┬function_name┬——┬,—0———————┬——————————►
                                └label————————┘  └,—com_area┘

►—,—class1—,—class2—┬——————————————————————————————┬—)——————————————►◄
                    └,—ctl_action—┬————————————————┤
                                  └,—msgid_list————┘
```

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, respectively, 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* have to 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 prior to 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 prior to 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.

See the *WebSphere Development Studio: ILE C/C++ Programmer’s Guide* for examples and more information about using the #pragma exception_handler directive.

hashome

C++

```
►►──#──pragma──hashome──(──className──┬────────────┬──)─────────────────────►◄
                                      └─AllInlines─┘
```

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 50.

implementation

C++

```
►►──#──pragma──implementation──(──string_literal──)──────────────────────►◄
```

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++

```
►►─#─pragma─info─(──┬─all──────────────────┬─)──────────────────────►◄
                    ├─none─────────────────┤
                    ├─restore──────────────┤
                    │  ┌────────────────┐  │
                    │  ▼                │  │
                    └───┬─nogroup─┬─,──┴──┘
                        └─group───┘
```

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
use	Check for unused auto or static variables
gen	List the general CHECKOUT messages

inline

C

```
►►—#—pragma—inline—(—function_name—)——————————————————————————►◄
```

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) parameter is not specified on the Create Module or Create Bound Program commands. 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_nam*e 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 the "Function Call Performance" in the *WebSphere Development Studio: ILE C/C++ Programmer's Guide* for more information on function inlining.

ishome

▶ C++

```
▶▶—#—pragma—ishome—(—className—)————————————————————————▶◀
```

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 46.

isolated_call

C++

```
▶▶──#──pragma──isolated_call──=──function────────────────────────────▶◀
```

Description

Lists a function that does not have or rely on side effects, other than those 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 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 above

Essentially, any change in the state of the run-time 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 run-time 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

```
►►─#─pragma─linkage─(──┬─program_name─┬──,─OS──┬──────────┬──)──────────►◄
                       └─typedef_name─┘         └─,─nowiden─┘
```

Description

Identifies a given function or function typedef as being an external program subject to OS/400 parameter passing conventions.

This pragma allows calls only to external programs. For information on making calls to bound procedures, see #pragma “argument” on page 25.

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 OS/400 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 OS/400 calling conventions.
nowiden	If specified, arguments are not widened before they are copied and passed.

Notes on Usage

This pragma lets an iSeries 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 OS/400 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 ▶ C++

```
►►—#—pragma—map—(—name1—,—"—name2—"—)——————————————————————————►◄
```

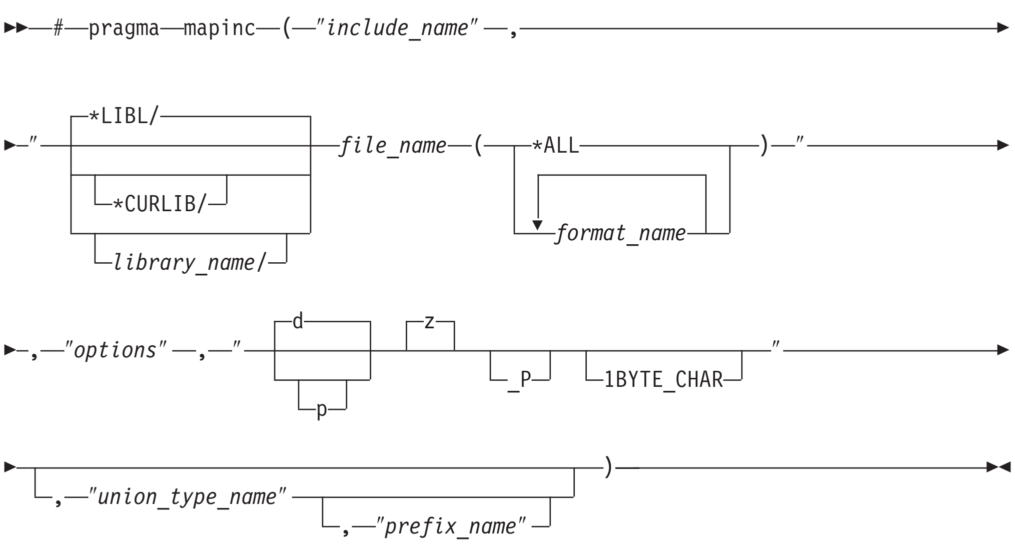
Description

Specifies that the compiler is to replace the external symbol (that is used in your C source) *name1* with the external symbol *name2*. Case significance is preserved only for those systems that support case distinction for external symbols.

The #pragma map directive supports library-qualified external program names. See "linkage" on page 52 for more information.

mapinc

C



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 on 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 *WebSphere Development Studio: ILE C/C++ Programmer's Guide* for more information about using the #pragma mapinc directive with externally described files.

margins

C

```
►►—#—pragma—margins—(—left margin—,—┬—right margin—┬—)————————————►◄
                                    └—*————————————┘
```

Description

Specifies the left and right margins to be used as the first and last column, respectively, 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 62 and "sequence" on page 75.

namemangling

C++

```
                                      ┌ansi───┐
▶▶──#──pragma──namemangling──(──┼compat─┼──┬─────────────┬────────────────▶◀
                                      ├v3─────┤   └,──num_chars─┘
                                      └v5─────┘
```

Description

Sets the maximum length for external symbol names generated from C++ source code.

Parameters

ansi	The name mangling scheme complies with the C++ standard. If you specify **ansi** but do not specify a size, the default maximum is 64000 characters.
compat	The name mangling scheme is the same as in versions of the compiler prior to V5R1M0. The default maximum is 255 characters. Use this scheme for compatibility with link modules created with earlier versions of the compiler.
v3	This option is the same as **compat**, described above.
v5	The name mangling scheme is the same as used in the V5R1M0 and V5R2M0 versions of the compiler. The default maximum is 64000 characters. Use this scheme for compatibility with link modules created with earlier versions of the compiler.
num_chars	Optionally specifies a maximum length for external symbol names generated from C++ source code.

Note on Usage

This pragma has effect only when the RTBND(*DEFAULT) compiler option is in effect.

noargv0

C

```
►►──#──pragma──noargv0──────────────────────────────────────────────►◄
```

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

```
►►──#──pragma──noinline──(──function_name──)──────────────────────────►◄
```

Description

Specifies that a function will not be inlined. The settings on the INLINE parameter of the Create Module or Create Bound Program commands 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 that is issued if it is not found at file scope.

nomargins

C

```
►►──#──pragma──nomargins──────────────────────────────────────────────────►◄
```

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 58.

nosequence

C

```
►►──#──pragma──nosequence──────────────────────────────────────────────►◄
```

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 75.

nosigtrunc

C

```
►►──#──pragma──nosigtrunc──────────────────────────────────────────────►◄
```

Description

Specifies that no exception is generated at run time 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 *WebSphere Development Studio: ILE C/C++ Programmer's Guide* for more information about packed decimal errors.

pack

C C++

```
►►─#─pragma─pack─(─┬────────────┬─)────────────────────────────────────►◄
                   ├─1──────────┤
                   ├─2──────────┤
                   ├─4──────────┤
                   ├─8──────────┤
                   ├─16─────────┤
                   ├─default────┤
                   ├─system─────┤
                   ├─pop────────┤
                   └─reset──────┘
```

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 Create Module or Create Bound Program commands to cause packing to be performed along specified boundaries. See "PACKSTRUCT" on page 127 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 iSeries 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.

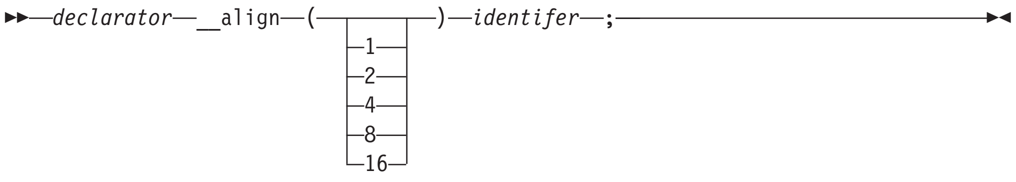
By default, all members use their natural alignment. Members cannot be aligned on values greater than their natural alignment. Char types can only be aligned along 1-byte boundaries. Short types can only be aligned along 1 or 2-byte boundaries, and int types can be aligned along on 1, 2, or 4-byte boundaries.

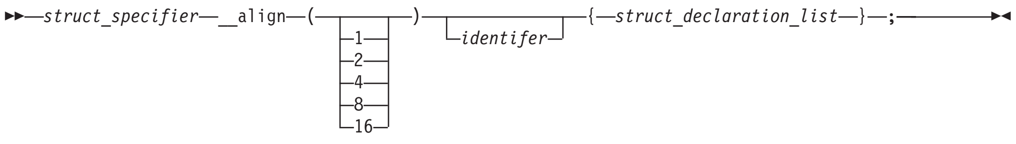
All 16-byte pointers will be aligned on a 16-byte boundary. **_Packed**, PACKSTRUCT, and #pragma pack cannot alter this. 8-byte teraspace pointers may have any alignment, although 8-byte alignment is preferred.

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.





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. It has the same effect as #pragma pack(1). The following are examples of legal and illegal usages of **_Packed**. In these examples, the keywords *struct*, *union*, and *class* can be used interchangeably.

```
_Packed class SomeClass { /* ... */ };        // OK
typedef _Packed union AnotherClass {} PUnion; // OK
typedef _Packed struct {} PAnonStruct;        // Illegal, struct must be named
_Packed SomeClass someObject;                 // Illegal, specifier _Packed must be
                                              // associated with class definition.
_Packed struct SomeStruct { };                // OK
_Packed union SomeUnion { };                  // OK
```

__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 would be determined according to the alignment rule on a specific platform. 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 should be **size_t**.

Examples

In the examples that follow, the words union and class could 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

```
►►──#──pragma──page──(──┬───┬──)──────────────────────────────────►◄
                        └─n─┘
```

Description

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

pagesize

C

```
►►──#──pragma──pagesize──(─┬───┬─)──────────────────────────────────────►◄
                           └─n─┘
```

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++

```
▶▶—#—pragma—pointer—(—typedef_name—,—pointer_type—)——————————————▶◀
```

Description

Allows the use of iSeries 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 *WebSphere Development Studio: ILE C/C++ Programmer's Guide* for more information about using iSeries pointers.

priority

C++

```
▶▶─#─pragma─priority─(─n─)──────────────────────────────────────────▶◀
```

Description

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

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 run-time 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 run-time 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

```
▶▶──#──pragma──sequence──(──left_column──,──┬─right_column─┬──)──────────────────▶◀
                                            └─*────────────┘
```

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 63 and “margins” on page 58.

strings

▶ C ▶ C++

```
►►──#──pragma──strings──(──┬─readonly──┬──)──────────────────────────────►◄
                           └─writeable─┘
```

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 option on the Create Module or Create Bound Program commands.

weak

C++

```
►►—#—pragma—weak—(—identifier——┬──────────────┬—)————————————————————————►◄
                               ├─=────────────┤
                               └─identifier2──┘
```

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
```

Chapter 4. Control Language Commands

This chapter discusses 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 ILE C/C++ compiler.

Table 1. 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 manual, 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.

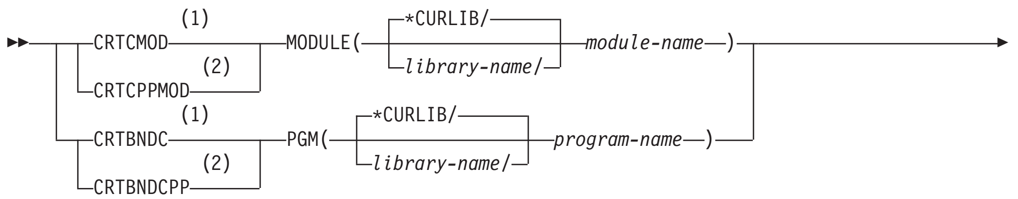
Language statements may contain punctuation marks, parentheses, arithmetic operators, or other such symbols. You must enter them exactly as shown in the syntax diagram.

You can also invoke the compiler and its options through the Qshell command line environment. For more information on Qshell command and option formats, see Chapter 5, "Using the ixlc Command to Invoke the C/C++ Compiler," on page 143.

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 86.

Syntax Diagram



```
                                                                 (5)
►─┬──────────────────────────────────────────────────────────────┬──────────────────►
  │                                     (2) (4)                   │
  │                           ┌─QCPPSRC──────────┐                │
  │                           │       (1) (3)    │                │
  │          ┌─*LIBL/───────┐ ├─QCSRC────────────┤                │
  └─SRCFILE(─┼──────────────┼─┼──────────────────┼─)──────────────┘
             ├─*CURLIB/─────┤ └─source-file-name─┘
             └─library-name/┘

►─┬──────────────────────────────┬─┬─────────────────────────────┬──────────────────►
  │             (7)              │ └─SRCSTMF(─┬───────────┬─)─────┘
  │        ┌─*PGM──────────┐     │            └─path-name─┘
  │        │        (6)    │     │
  │        ├─*MODULE───────┤     │
  └─SRCMBR(─┼───────────────┼─)───┘
           └─member-name───┘

►─┬────────────────────────────────────┬─────────────────────────────────────────────►
  │      ┌─*SRCMBRTXT──────────┐       │
  └─TEXT(─┼─*BLANK──────────────┼─)─────┘
         └─'──description──'───┘

►─┬──────────────────────────────────────────────────────────────────────────────┬───►
  │         ┌─*NONE─────┐                                                          │
  └─OUTPUT(─┼─*PRINT────┼─┬───────────────────────┬─┬───────────────────────────┬─)─┘
            └─filename──┘ │        ┌─*BLANK─┐     │ │           ┌─*BLANK────┐    │
                          └─TITLE──┴─title──┴─────┘ └─SUBTITLE──┴─subtitle──┴────┘

►─┬─────────────────────────────────┬─┬─────────────────────────────────────┬───────►
  └─OPTION(──┤ OPTION Details ├──)──┘ └─CHECKOUT(──┤ CHECKOUT Details ├──)──┘

►─┬─────────────────────────┬─┬─────────────────────────────────┬───────────────────►
  │           ┌─10─┐        │ └─INLINE(──┤ INLINE Details ├──)──┘
  └─OPTIMIZE(─┼─20─┼──)─────┘
              ├─30─┤
              └─40─┘

►─┬──────────────────────────────────────────────┬─┬───────────────────────────────┬►
  │          (8)        ┌─*NOKEEPILDTA─┐         │ │          ┌─*NONE───┐           │
  └─MODCRTOPT(──────────┴─*KEEPILDTA───┴─)───────┘ └─DBGVIEW(──┼─*ALL────┼──)───────┘
                                                              ├─*STMT───┤
                                                              ├─*SOURCE─┤
                                                              └─*LIST───┘

►─┬─────────────────────────────────────────────────┬─┬──────────────────────────────┬►
  │        ┌─*NONE──────────────────────────────┐   │ │          ┌─*EXTENDED──────┐   │
  │        │    ┌─────────────────────────────┐ │   │ └─LANGLVL(──┼─*ANSI──────────┼─)─┘
  │        │    ▼                             │ │   │            │          (2)   │
  └─DEFINE(─┴────┬─'──name──'───────────────┬─┴─┴─)─┘            └─*LEGACY────────┘
                └─'──name──=──value──'─────┘

►─┬──────────────────────────────┬──────────────────────────────────────────────────►
  │       ┌─*ANSI──────────┐     │
  └─ALIAS(─┼─*NOANSI────────┼─)───┘
          ├─*ADDRTAKEN─────┤
          ├─*NOADDRTAKEN───┤
          ├─*ALLPTRS───────┤
          ├─*NOALLPTRS─────┤
          ├─*TYPEPTR───────┤
          └─*NOTYPEPTR─────┘
```

```
►─┬──────────────────────────────────────────────────────────────────────────────────┬────────────────────────►
  │                       (4)                                                        │
  │             ┌─*IFS64IO──────┐                                                    │
  │             │         (3)   │                              (1)                   │
  │             ├─*NOIFSIO──────┤  ┌─*NOASYNCSIGNAL────────────┐                     │
  └─SYSIFCOPT(──┼───────────────┼──┼───────────────────────────┼──)─┘
                └─*IFSIO────────┘  │                     (1)   │
                                   └─*ASYNCSIGNAL──────────────┘

►─┬─────────────────────────────────────┬──┬──────────────────────┬──────────────────────────────────────────►
  │                ┌─*LOCALE─────────┐  │  │         ┌─0──┐        │
  └─LOCALETYPE(────┼─*LOCALEUCS2─────┼──)─┘  └─FLAG(──┼─10─┼──)─────┘
                   │        (1)      │                ├─20─┤
                   ├─*CLD────────────┤                └─30─┘
                   └─*LOCALEUTF──────┘

►─┬─────────────────────────────────────┬──┬───────────────────────────┬──────────────────────────────────────►
  │         ┌─*NOMAX────┐   ┌─30─┐      │  │           ┌─*YES─┐         │
  └─MSGLMT(─┴─0 32767───┴───┼─0──┼──)───┘  └─REPLACE(──┴─*NO──┴───)─────┘
                            ├─10─┤
                            └─20─┘

►─┬────────────────────────────────┬──┬──────────────────────────────────────────────┬────────────────────────►
  │       (9)     ┌─*USER──┐       │  │       ┌─*LIBCRTAUT─────────────────┐         │
  └─USRPRF(───────┴─*OWNER─┴───)───┘  └─AUT(──┼─*CHANGE────────────────────┼──)──────┘
                                              ├─*USE───────────────────────┤
                                              ├─*ALL───────────────────────┤
                                              ├─*EXCLUDE───────────────────┤
                                              └─authorization-list-name────┘

►─┬────────────────────────────────────┬──────────────────────────────────────────────────────────────────────►
  │          ┌─*CURRENT──────┐         │
  └─TGTRLS(──┼─*PRV──────────┼───)─────┘
             └─release-level─┘

►─┬──────────────────────────────────────────────────────────────┬────────────────────────────────────────────►
  │             ┌─*PEP───────────────────────────────────┐       │
  └─ENBPFRCOL(──┼────────────────────────────────────────┼──)────┘
                └─┬─*ENTRYEXIT─┬──────┬─*ALLPRC──┬───────┘
                  └─*FULL──────┘      └─*NONLEAF─┘

►─┬──────────────────────────────────────────────────────────┬──┬────────────────────────────┬────────────────►
  │         ┌─*SETFPCA────┐   ┌─*NOSTRDONLY─┐                │  │          ┌─*NOCOL─┐        │
  └─PFROPT(─┼─────────────┼───┼─────────────┼───)────────────┘  └─PRFDTA(──┴─*COL───┴──)─────┘
            └─*NOSETFPCA──┘   └─*STRDONLY───┘

►─┬────────────────────────────────────────────────────┬──┬──────────────────────────────────┬────────────────►
  │             ┌─*NO────────────────────────────┐     │  │          ┌─*SNGLVL────┐          │
  │             │                ┌─*NOTSIFC─┐    │     │  └─STGMDL(──┼─*TERASPACE─┼──)───────┘
  └─TERASPACE(──┴───────*YES─────┼──────────┼────┴──)──┘             └─*INHERIT───┘
                                 └─*TSIFC───┘

►─┬────────────────────────────────┬──┬─────────────────────────────────┬─────────────────────────────────────►
  │         ┌─*P128──┐             │  │         ┌─*DEFAULT─┐            │
  └─DTAMDL(─┴─*LLP64─┴────)────────┘  └─RTBND(──┴─*LLP64───┴─────)──────┘
```

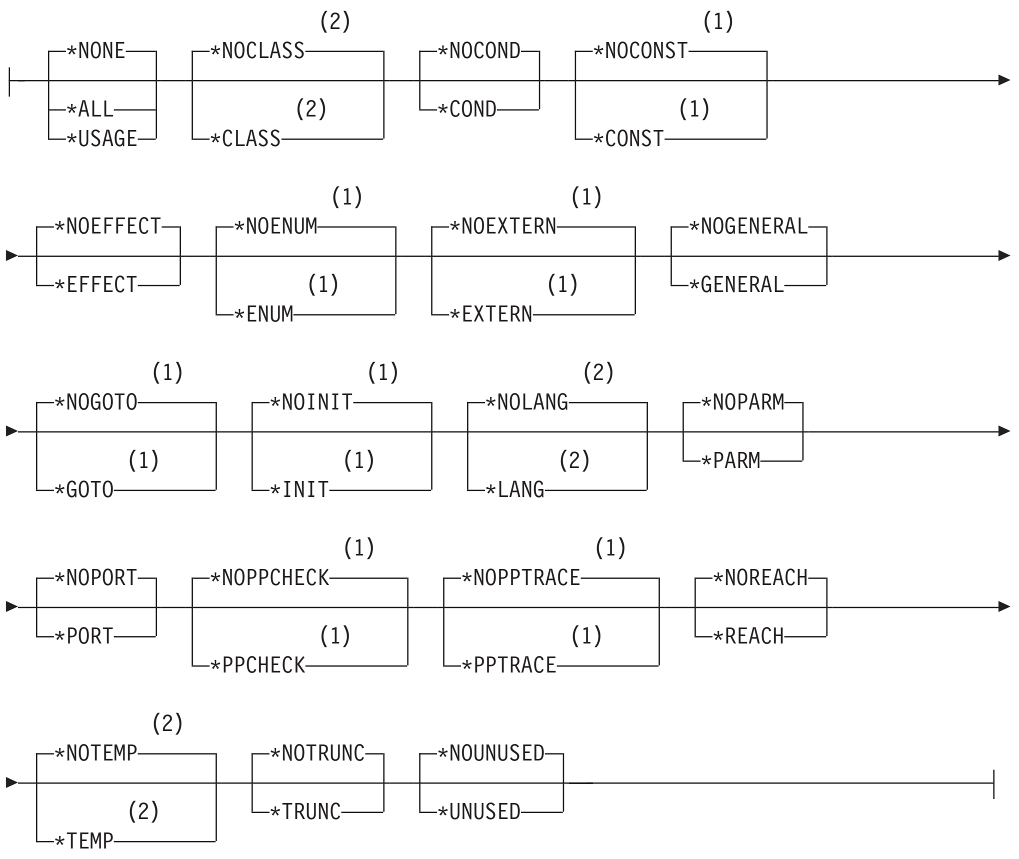
```
►──┬──────────────────────────────────┬──┬─────────────────────────┬──────────────────────►
   │                ┌─*NATURAL─┐      │  │        ┌─*SMALL─┐       │
   └─PACKSTRUCT(────┼─1────────┼──)───┘  └─ENUM(──┼─1──────┼──)────┘
                    ├─2────────┤                  ├─2──────┤
                    ├─4────────┤                  ├─4──────┤
                    ├─8────────┤                  └─*INT───┘
                    └─16───────┘

►──┬────────────────────────────────┬────────────────────────────────────────────────────►
   │             ┌─*NODEP─────┐     │
   └─MAKEDEP(────┴─file-name──┴──)──┘

►──┬──────────────────────────────────────────────────────────────┬──────────────────────►
   │              ┌─*NONE──────────────────────────────────────┐  │
   └─PPGENOPT(────┼────────────────────────────────────────────┼──)─┘
                  ├─*DFT───────────────────────────────────────┤
                  │                          ┌─*GENLINE───┐    │
                  ├──┬─*RMVCOMMENT───┬───────┼────────────┼────┤
                  │  └─*NORMVCOMMENT─┘       └─*NOGENLINE─┘    │
                  │                    ┌─*RMVCOMMENT───┐       │
                  └──┬─*GENLINE───┬────┼───────────────┼───────┘
                     └─*NOGENLINE─┘    └─*NORMVCOMMENT─┘

►──┬──────────────────────────────────────────────────────────────┬──────────────────────►
   │            (8)      ┌─*CURLIB/───────┐                       │
   └─PPSRCFILE(──────────┴─library-name/──┴──file-name──)─────────┘

►──┬──────────────────────────────────────┬──┬──────────────────────────────────────┬───►
   │           (8)      ┌─*MODULE────┐    │  │            (8)                       │
   └─PPSRCMBR(──────────┴─membername─┴──)─┘  └─PPSRCSTMF(──────────┬────────────┬─)─┘
                                                                   ├─pathname───┤
                                                                   └─*SRCSTMF───┘

►──┬────────────────────────────────────────────┬────────────────────────────────────────►
   │          ┌─*NONE────────────────────┐      │
   │          │   ┌──────────────────┐   │      │
   │          │   ▼                  │   │      │
   └─INCDIR(──┴───────directory-name─┴───┴──)───┘

►──┬────────────────────────────────────────────────────────┬────────────────────────────►
   │         ┌─*NONE─────────────────────────────────────┐  │
   └─CSOPT(──┴─'──compiler-service-options-string──'───┴──)─┘

►──┬────────────────────────────────────────────────────────────────┬────────────────────►
   │          ┌─*NONE──────────────────────────────────────────┐    │
   └─LICOPT(──┴─'──Licensed-Internal-Code-Options-String──'────┴──)─┘

►──┬─────────────────────────────────┬──┬───────────────────────────────────────────────┬─►
   │            ┌─*UNSIGNED─┐        │  │             ┌─*SOURCE──────────────────────┐  │
   └─DFTCHAR(───┴─*SIGNED───┴──)─────┘  └─TGTCCSID(───┼─*JOB─────────────────────────┼)─┘
                                                      ├─*HEX─────────────────────────┤
                                                      └─coded-character-set-identifier┘

►──┬──────────────────────────────────────────────────────────────┬──────────────────────►
   │           (2) (8) (10)       ┌─*NONE──────────────────┐      │
   └─TEMPLATE(────────────────────┴─┤ TEMPLATE Details ├───┴──)───┘
```

```
►─┬───────────────────────────────────────────────────────────┬────────────────────────────────►
  │            (2) (8) (10)    ┌─*NONE──────────────┐          │
  └─TMPLREG(──────────────────┼─*DFT───────────────┼──)───────┘
                               └─'──path-name──'────┘

►─┬──────────────────────────────────────────────┬─────────────────────────────────────────────►◄
  │            (2) (10)       ┌─*YES─┐            │
  └─WEAKTMPL(────────────────┴─*NO──┴──)────────┘
```

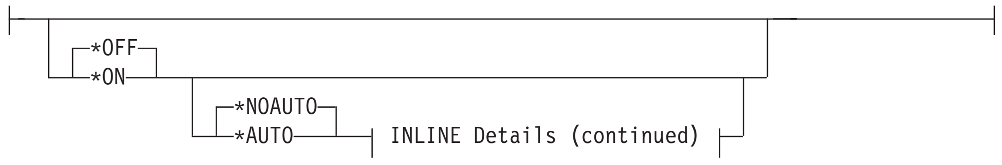
OPTION Details:

```
            (1)                    (2)                  (1)
   ┌─*NOAGR──────────┐  ┌─*NOBITSIGN──────────┐  ┌─*DIGRAPH──────────────┐  ┌─*NOEVENTF─┐
├──┼─────────────────┼──┼─────────────────────┼──┼───────────────────────┼──┼───────────┼──►
   │         (1)     │  │             (2)     │  │               (1)     │  └─*EVENTF───┘
   └─*AGR────────────┘  └─*BITSIGN────────────┘  └─*NODIGRAPH────────────┘

                                         (8)
   ┌─*NOEXPMAC─┐  ┌─*NOFULL─┐  ┌─*GEN───────────┐  ┌─*NOINCDIRFIRST─┐
►──┼───────────┼──┼─────────┼──┼────────────────┼──┼────────────────┼──────────────────────────►
   └─*EXPMAC───┘  └─*FULL───┘  │          (8)   │  └─*INCDIRFIRST───┘
                               └─*NOGEN─────────┘

                               (2)                  (2)
   ┌─*LOGMSG───┐  ┌─*LONGLONG──────────┐  ┌─*NORTTI──────────┐
►──┼───────────┼──┼────────────────────┼──┼──────────────────┼─────────────────────────────────►
   └─*NOLOGMSG─┘  │             (2)    │  │            (2)   │
                  └─*NOLONGLONG────────┘  ├─*RTTIALL─────────┤
                                          │            (2)   │
                                          ├─*RTTITYPE────────┤
                                          │            (2)   │
                                          └─*RTTICAST────────┘

                (1) (8)                (1)
   ┌─*NOPPONLY──────────┐  ┌─*NOSECLVL─────────┐  ┌─*NOSHOWINC─┐
►──┼────────────────────┼──┼───────────────────┼──┼────────────┼───────────────────────────────►
   │          (1) (8)   │  │            (1)    │  └─*SHOWINC───┘
   └─*PPONLY────────────┘  └─*SECLVL───────────┘

                (1)
   ┌─*NOSHOWSKP───────┐  ┌─*SHOWSRC───┐  ┌─*NOSHOWSYS─┐  ┌─*NOSHOWUSR─┐
►──┼──────────────────┼──┼────────────┼──┼────────────┼──┼────────────┼────────────────────────►
   │          (1)     │  └─*NOSHOWSRC─┘  └─*SHOWSYS───┘  └─*SHOWUSR───┘
   └─*SHOWSKP─────────┘

                                                      (1)
   ┌─*STDINC───┐  ┌─*NOSTDLOGMSG─┐  ┌─*NOSTRUCREF─────────┐  ┌─*NOSYSINCPATH─┐
►──┼───────────┼──┼──────────────┼──┼─────────────────────┼──┼───────────────┼─────────────────►
   └─*NOSTDINC─┘  └─*STDLOGMSG───┘  │            (1)      │  └─*SYSINCPATH───┘
                                    └─*STRUCREF───────────┘

   ┌─*NOXREF─┐  ┌─*NOXREFREF─┐
►──┼─────────┼──┼────────────┼─────────────────────────────────────────────────────────────────┤
   └─*XREF───┘  └─*XREFREF───┘
```

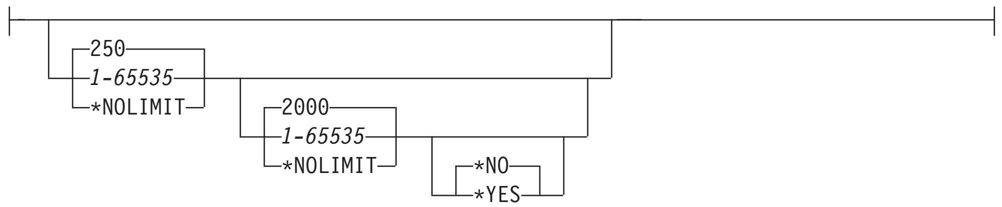
CHECKOUT Details:



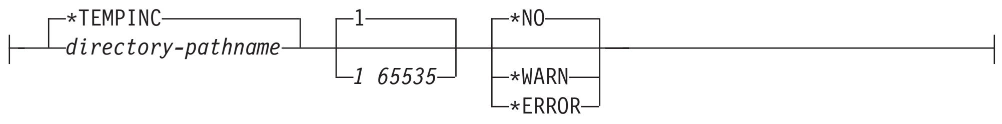
INLINE Details:



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	All parameters preceding this point can be specified positionally
6	Create Module command only
7	Create Bound Program command only
8	Create Module command only
9	Create Bound Program command only
10	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 will have 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.

```
                ┌─*CURLIB/──────┐
├──MODULE(──────┼───────────────┼──module-name──)──────────────────────────────┤
                └─library-name/─┘
```

<u>*CURLIB</u>
: 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.

```
             ┌─*CURLIB/──────┐
├──PGM(──────┼───────────────┼──program-name──)────────────────────────────────┤
             └─library-name/─┘
```

<u>*CURLIB</u>
: 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.

```
├──┬─────────────────────────────────────────────────────────────┬──┤
   │                                    (3) (4)                  │
   │                           ┌─QCPPSRC──────────┐              │
   │                           │          (1) (2) │              │
   │            ┌─*LIBL/───────┐├─QCSRC────────────┤              │
   └─SRCFILE(───┼──────────────┼┼──────────────────┼──)───────────┘
                ├─*CURLIB/─────┤└─source-file-name─┘
                └─library-name/┘
```

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.

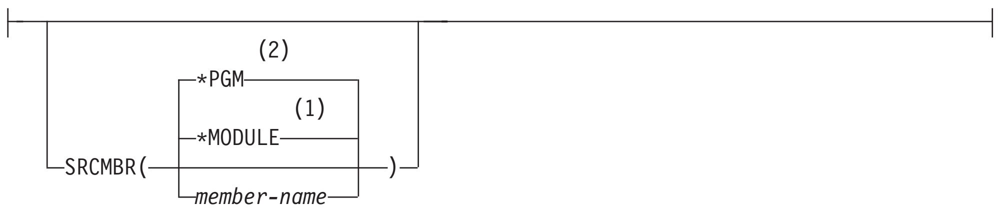
QCSRC C
The default name for the source physical file that contains the member with the ILE C source code that you want to compile.

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.



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(──┬───────────┬──)─┘
               └─path-name─┘
```

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.

Notes:

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:
   - INCDIR( )
   - OPTION(*INCDIRFIRST)
   - 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.

```
├──┬────────────────────────────────┬──────────────────────────────────────┤
   │          ┌─*SRCMBRTXT──────┐    │
   └─TEXT(────┼─*BLANK──────────┼──)─┘
              └─'─description─'─┘
```

***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.

```
├──┬────────────────────────────────────────────────────────────────────────────┬──────┤
   │           ┌─*NONE────┐                                                      │
   └─OUTPUT(───┼─*PRINT───┼──┬──────────────────────┬──┬────────────────────────────┬──)─┘
               └─filename─┘  │          ┌─*BLANK─┐  │  │             ┌─*BLANK────┐  │
                             └─TITLE────┴─title──┴──┘  └─SUBTITLE────┴─subtitle──┴──┘
```

***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.

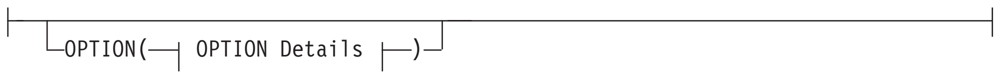
SUBTITLE
Specifies the subtitle for the compiler listing. Possible SUBTITLE values are:

***BLANK**
No title is generated.

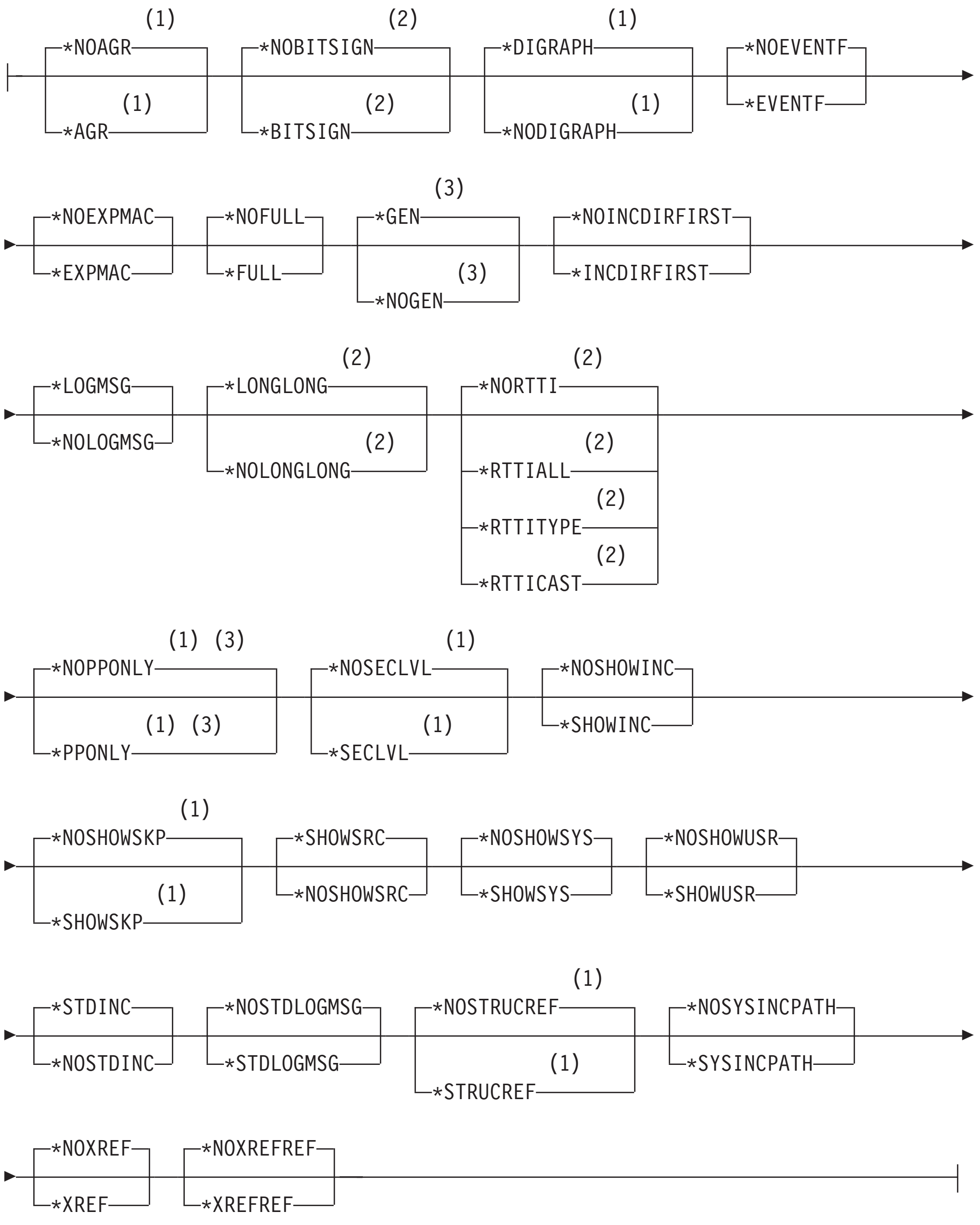
subtitle
Specify a subtitle string (maximum 98 characters) for the listing file.

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 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 may 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 shows all compiler-ouput information in the listing or in the debug listing view.

***FULL**
Shows all compiler-ouput 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 Run-Time 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.

***RTTICAST** C++
The compiler generates the information needed for the RTTI dynamic_cast operator, but the information for the typeid operator is not generated.

***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) may 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 and/or *NOSHOWSYS after *SHOWINC.

<u>*NOSHOWSKP</u> 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 or not the preprocessor has skipped them. OUTPUT(*PRINT) or DBGVIEW(*ALL or *LIST) must be specified.

<u>*SHOWSRC</u>

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,*SHOWSYS and *SHOWSKP listing options can override this setting if specifed after the*NOSHOWSRC option.

<u>*NOSHOWSYS</u>

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 (< >).

<u>*NOSHOWUSR</u>

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.

<u>*STDINC</u>

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 will produce 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. This 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.

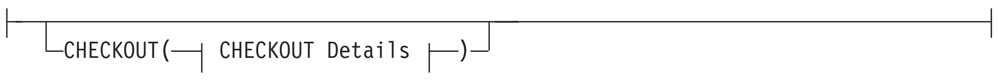
***XREFREF**
Generates the cross-reference table, including only referenced identifers and variables in the source code, together with the line number in which they appear. An OUTPUT option must be specified.

The *XREF option overrides the *XREFREF option.

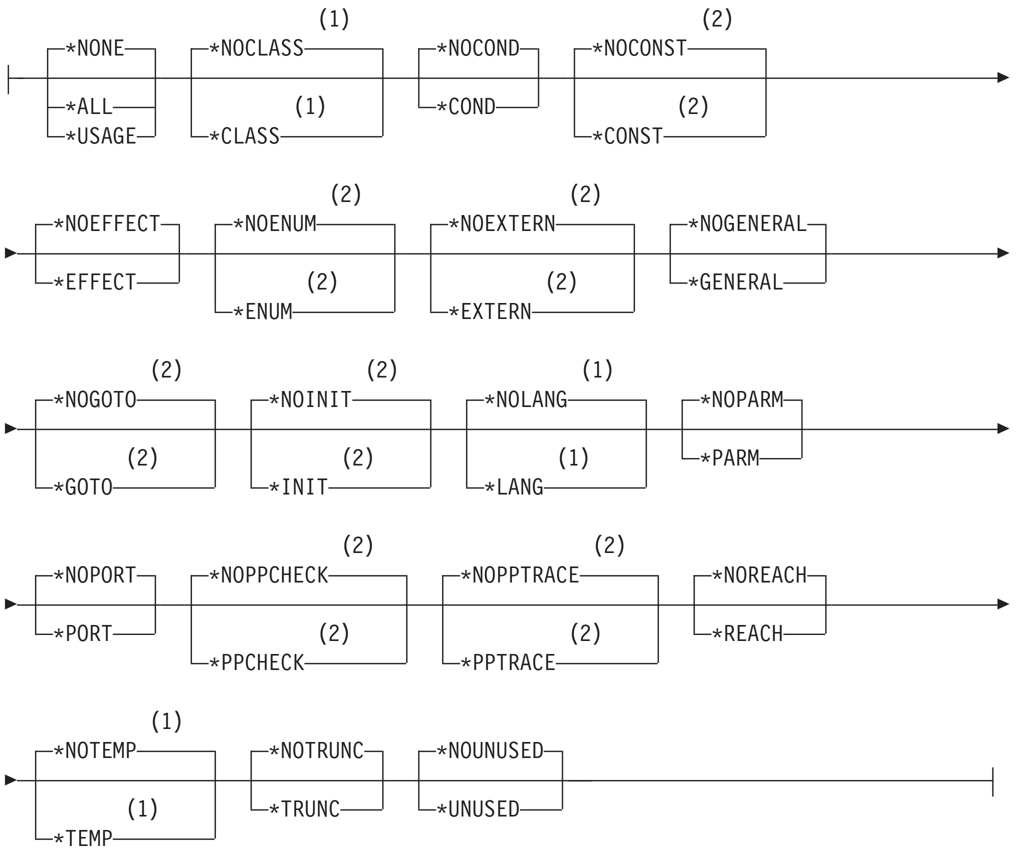
CHECKOUT

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 Details:



Notes:

1 C++ compiler only

2 C compiler only

The possible options are:

<u>*NONE</u>
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 C
: Default setting. Does not list the occurrence and usage of goto statements.

*GOTO C
: 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** C
Default setting. Does not list the tracing of include files by the preprocessor.

***PPTRACE** C
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.

```
├──┬──────────────────────────┬──────────────────────────────────────────────┤
   │              ┌─10─┐      │
   └─OPTIMIZE(────┼─20─┼───)──┘
                  ├─30─┤
                  └─40─┘
```

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

20 Some optimization is performed on the code.

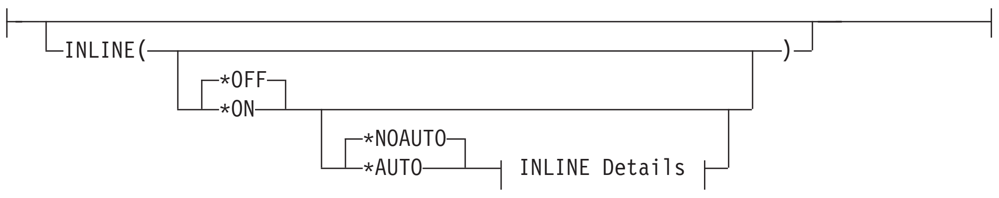
30 Full optimization is performed on the generated code.

40 All optimizations done at level 30 are performed on the generated code. In addition, code is eliminated from procedure prologue and epilogue routines that enable 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.

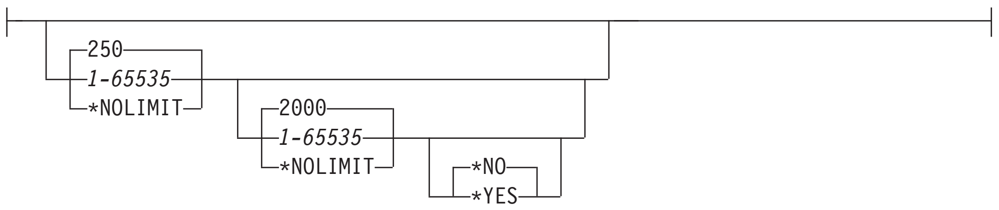
INLINE

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 Details (continued):



The possible INLINE options are:

Inliner
Specifies whether or not 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 or not the inliner should attempt to automatically inline functions depending on their Threshold and Limit.

***NOAUTO**
Specifies that only the functions that have been specified with the #pragma inline directive should be considered candidates for inlining. This is a default.

***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.

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 or not 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.

```
├──┬──────────────────────────────────────────┬──────────────────────────────────────┤
   │             (1)     ┌─*NOKEEPILDTA─┐     │
   └─MODCRTOPT(──────────┼──────────────┼──)──┘
                         └─*KEEPILDTA───┘
```

Notes:

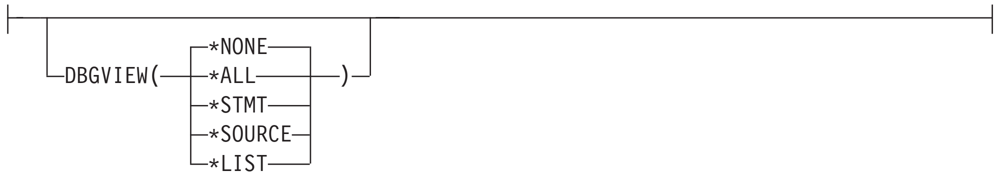
1 Create Module command only

***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.



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.

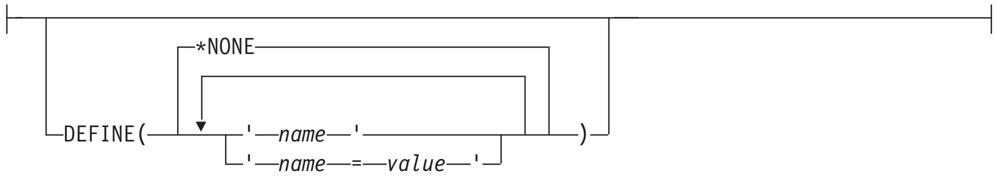
***SOURCE**
Generates the source view for debugging the compiled object. The OPTION(*NOSHOWINC, *SHOWINC, *SHOWSYS, *SHOWUSR) determines the content of the source view that is created.

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.

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').



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 library function prototypes are included when the source is compiled. When no LANGLVL is specified, the language level defaults to *EXTENDED.

```
├──┬─────────────────────────────────┬──────────────────────────────────────────┤
   │              ┌─*EXTENDED──┐     │
   └─LANGLVL(─────┼─*ANSI──────┼──)──┘
                  │       (1)  │
                  └─*LEGACY────┘
```

Notes:

1 C++ compiler only

<u>*EXTENDED</u>
: 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.

ALIAS

Specifies the aliasing assertion to be applied in the module created.

```
├──┬──────────────────────────────┬──────────────────────────────────────────────┤
   └─ALIAS(──┬─*ANSI────────┬──)─┘
             ├─*NOANSI──────┤
             ├─*ADDRTAKEN───┤
             ├─*NOADDRTAKEN─┤
             ├─*ALLPTRS─────┤
             ├─*NOALLPTRS───┤
             ├─*TYPEPTR─────┤
             └─*NOTYPEPTR───┘
```

<u>*ANSI</u>
: Default setting. The module or program created will only allow pointers to point to an object of the same type.

***NOANSI**
: The module or program created will not use the *ANSI aliasing rules.

***ADDRTAKEN**
: The module or program created will have its class of variables disjoint from pointers unless their address is taken.

***NOADDRTAKEN**
: The module or program created will not use the *ADDRTAKEN aliasing rules.

***ALLPTRS**
: The module or program created will not allow any two pointers to be aliased.

***NOALLPTRS**
: The module or program created will not use the *ALLPTRS aliasing rules.

***TYPEPTR**
: The module or program created will not allow any two pointers of different types to be aliased.

***NOTYPEPTR**
: The module or program created will not use the *TYPEPTR aliasing rules.

SYSIFCOPT

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

```
├──┬─────────────────────────────────────────────────────────────┬──┤
   │                   (2)                                        │
   │         ┌─*IFS64IO──────┐                                    │
   │         │         (1)   │                  (3)               │
   │         ├─*NOIFSIO──────┤   ┌─*NOASYNCSIGNAL──┐              │
   └─SYSIFCOPT(─┼───────────────┼───┼─────────────────┼─)─────────┘
             │               │   │             (3)  │
             └─*IFSIO────────┘   └─*ASYNCSIGNAL─────┘
```

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 iSeries 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**
Default setting. Does not enable run-time mapping of synchronous signalling functions to asynchronous signalling functions.

***ASYNCSIGNAL**
Enable run-time mapping of synchronous signalling functions to asynchronous signalling functions. Specifying this option causes C run-time environment to map the synchronous **signal()** and **raise()** functions to the asynchronous **sigaction()** and **kill()** functions respectively.

LOCALETYPE

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

```
├─┬─────────────────────────────────────┬──────────────────────────────────────────┤
  │              ┌─*LOCALE──────┐       │
  └─LOCALETYPE(──┼─*LOCALEUCS2──┼──)────┘
                 │     (1)      │
                 ├─*CLD─────────┤
                 └─*LOCALEUTF───┘
```

Notes:

1 C compiler only

<u>*LOCALE</u>
Default setting. Objects compiled with this option use the locale support provided with the ILE C/C++ compiler and run time, using locale objects of type *LOCALE. This option is only valid for programs that run on V3R7 and later releases of the OS/400® 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 run time, 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.

```
├──┬─────────────────────┬─────────────────────────────────────────────────────┤
   │                     │
   │         ┌─0──┐      │
   └─FLAG(───┼─10─┼──)───┘
             ├─20─┤
             └─30─┘
```

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.

```
├──┬────────────────────────────────────┬──────────────────────────────────────────┤
   │           ┌─*NOMAX──┐   ┌─30─┐     │
   └─MSGLMT(───┴─0 32767─┴───┼─0──┼──)─┘
                             ├─10─┤
                             └─20─┘
```

***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.

```
├──┬──────────────────────────┬──────────────────────────────────────────┤
   │             ┌─*YES─┐     │
   └─REPLACE(────┴─*NO──┴───)─┘
```

***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.

```
├──┬────────────────────────────────┬──────────────────────────────────────────┤
   │         (1)    ┌─*USER──┐      │
   └─USRPRF(────────┴─*OWNER─┴───)──┘
```

Notes:

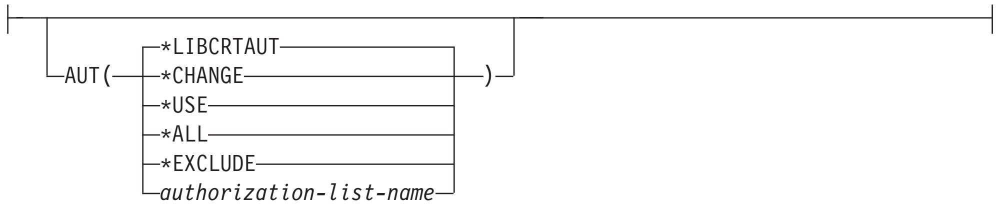
1 Create Bound Program command only

***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.



***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.

```
├──┬──────────────────────────────────┬──────────────────────────────────────────┤
   │           ┌─*CURRENT──────┐      │
   └─TGTRLS(───┼─*PRV──────────┼──)───┘
               └─release-level─┘
```

***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 98
- "OPTION" on page 92
- "OUTPUT" on page 91
- "PRFDTA" on page 120

The following options are ignored completely when compiling for an operating system release earlier than V5R1M0:
- "CSOPT" on page 135
- "DFTCHAR" on page 137
- "DTAMDL" on page 125
- "ENUM" on page 128
- "INCDIR" on page 134
- "LICOPT" on page 136
- "MAKEDEP" on page 129
- "PACKSTRUCT" on page 127

- "PPGENOPT" on page 130
- "STGMDL" on page 124
- "TGTCCSID" on page 138

ENBPFRCOL

Specifies whether performance data measurement code should be generated in the object. The collected data that can be used by the system performance tool to profile an application's performance. Generating performance measurement code in an object will result in slightly larger objects and may affect performance.

```
├──┬─────────────────────────────────────────────────┬──────────────────────────┤
   │                ┌─*PEP────────────────────────┐   │
   └─ENBPFRCOL(─────┼─────────────────────────────┼─)─┘
                    └─┬─*ENTRYEXIT─┬──┬─*ALLPRC──┬┘
                      └─*FULL──────┘  └─*NONLEAF─┘
```

<u>*PEP</u>
Default setting. Performance statistics are only gathered on the entry and exit of the program entry procedure. 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 routine.

This choice is useful if you want to capture information on routines that call other routines 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 routine.

This choice is useful if you want to capture information on all routines. 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 will charge their use of resources against your application. This would make 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 includes leaf procedures. Also, statistics are gathered before and after each call to an external procedure.

Use this option if your application will call other objects that were not compiled with either *PEP, *ENTRYEXIT or *FULL. This option allows the performance tools to distinguish between resources that are used by your application and those 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 will be collected for this object. Use this parameter when no performance information is needed, and a smaller object size is desired.

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.

```
├──┬────────────────────────────────────────────────────────┬──────────────────────────────────┤
   │              ┌─*SETFPCA────┐    ┌─*NOSTRDONLY─┐         │
   └─PFROPT(──────┼─────────────┼────┼─────────────┼────)────┘
                  └─*NOSETFPCA──┘    └─*STRDONLY───┘
```

***SETFPCA**
Default setting. Causes the compiler to set the floating-point computational attributes to achieve the ANSI semantics for floating-point computations.

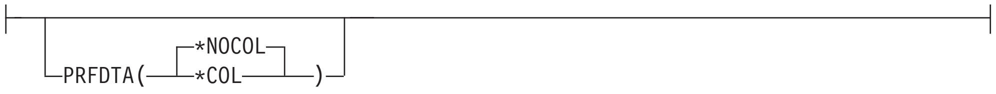
***NOSETFPCA**
No computational attributes will be set. This option is used when the object being created does not have any floating-point computations in it.

***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.



Note: You cannot profile a stand-alone *MODULE object.

<u>*NOCOL</u>
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.

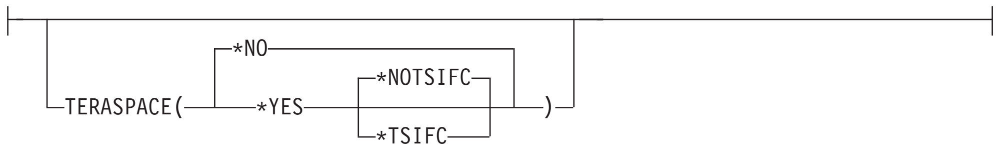
***COL**
The collection of profiling data is enabled. The module will collect profiling data when it is included in a program or service program object.

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.



***NO**
Default setting. The created object cannot recognize teraspace storage addresses.

***YES**
The created object can handle teraspace storage addresses, including parameters passed from other teraspace-enabled programs and service programs.

***NOTSIFC**
The compiler will not use teraspace versions of storage functions, such as malloc( ) or shmat( ). This 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 125) and STGMDL (see page 124) 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.

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.
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
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 should specify the following combination of options:

```
TERASPACE(*YES *TSIFC) STGMDL(*TERASPACE) DTAMDL(*LLP64)
```

For more information about teraspace storage, see *Using Teraspace* in *WebSphere Development Studio: ILE C/C++ Programmer's Guide*, and *Teraspace and single-level store* in *ILE Concepts*.

STGMDL

Specifies the type of storage (static and automatic) that the module object will use.

```
├──┬───────────────────────────────────┬──────────────────────────────────────┤
   │          ┌─*SNGLVL────┐           │
   └─STGMDL(──┼─*TERASPACE─┼──)────────┘
              └─*INHERIT───┘
```

***SNGLVL**
Default setting. The module or program will use the traditional single level storage model. Static and automatic storage for the object are allocated from single-level store, and can only be accessed using 16-byte pointers. The module may optionally access teraspace dynamic storage if the TERASPACE(*YES) option is specified.

***TERASPACE**
The module or program will use the teraspace storage model. This is a new storage model that provides up to a 1-terabyte local address space for a single job. Static and automatic storage for the object are 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 will depend on the type of storage required by the caller.

Use of STGMDL(*TERASPACE) or STGMDL(*INHERIT) together with TERASPACE(*NO) will be flagged as an error by the compiler, and compilation will stop.

The STGMDL(*TERASPACE) and STGMDL(*INHERIT) settings are ignored if used together with the TGTRLS compiler option and a target release earlier than V5R1M0 is specified.

For more information about valid combinations for the STGMDL, TERASPACE, and DTAMDL compiler options, see "TERASPACE" on page 121.

For more information about the types of storage available on iSeries systems, see *Teraspace and single-level store* in *ILE Concepts*.

DTAMDL

Specifies how pointer types will be 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.

```
├──┬────────────────────────────┬──────────────────────────────────────────┤
   │            ┌─*P128──┐      │
   └─DTAMDL(────┴─*LLP64─┴───)──┘
```

P128
: Default setting. The size of int, long, and pointer data types are 4, 4, and 16 bytes respectively.

***LLP64**
: The size of int, long, and pointer data types are 4, 4, and 8 bytes respectively, and the compiler will define the macro __LLP64_IFC__.

Use of DTAMDL(*LLP64) together with TERASPACE(*NO) will be flagged as an error by the compiler, and compilation will stop.

DTAMDL(*LLP64) is ignored if the TGTRLS compiler option specifies a target release earlier than V5R1M0.

See pragma “datamodel” on page 32 for more information.

For more information about valid combinations for the STGMDL, TERASPACE, and DTAMDL compiler options, see “TERASPACE” on page 121.

RTBND

Specifies the run-time binding directory for the object created.

```
├──┬──────────────────────────────┬──────────────────────────────────────────┤
   │            ┌─*DEFAULT─┐      │
   └─RTBND(─────┴─*LLP64───┴──)───┘
```

***DEFAULT**
: Default setting. The object created will use the default binding directory.

***LLP64**
: The object created will use the 64-bit run-time binding directory and the compiler will define the macro __LLP64_RTBND__. This suboption is valid only when TGTRLS(*CURRENT) is in effect, otherwise the compiler will issue an error message and stop the compilation.

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, but not for the structures themselves.

If the data types are by default packed along boundaries smaller than those 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 will be aligned on a 16-byte boundary. PACKSTRUCT, _Packed, and #pragma pack cannot alter this.
- 8-byte pointers can have any alignment, but 8-byte alignment is preferred.

For more information on packing and alignment, see pragma “pack” on page 65.

```
├──┬──────────────────────────────────┬────────────────────────────────────────────────┤
   │                ┌─*NATURAL─┐      │
   └─PACKSTRUCT(────┼─1────────┼───)──┘
                    ├─2────────┤
                    ├─4────────┤
                    ├─8────────┤
                    └─16───────┘
```

***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.

```
├──┬───────────────────────────┬──────────────────────────────────────────────────┤
   │          ┌─*SMALL─┐       │
   └─ENUM(──┼─1──────┼──)──┘
              ├─2──────┤
              ├─4──────┤
              └─*INT───┘
```

<u>*SMALL</u>
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.

```
├──┬────────────────────────────┬─────────────────────────────────────────────┤
   │           ┌─*NODEP────┐    │
   └─MAKEDEP(──┴─file-name─┴──)─┘
```

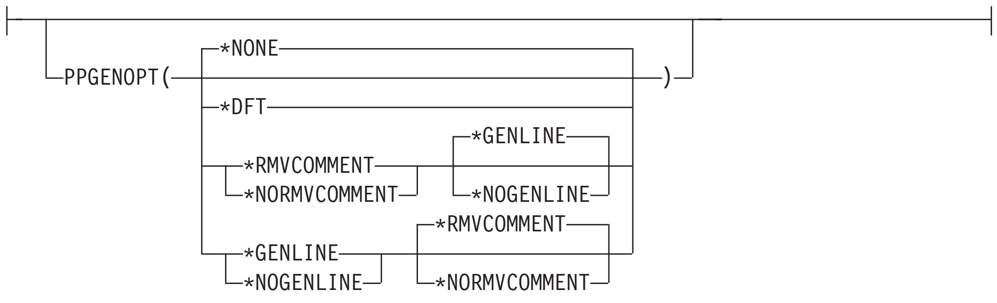
***NODEP**
: Default setting. The option is disabled and no file is created.

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.



***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:
   - PPSRCFILE and PPSRCMBR
   - PPSRCSTMF and SRCSTMF
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.
4. The PPGENOPT compiler option is ignored if the TGTRLS compiler option specifies a target release earlier than V5R1M0.

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.

```
├──┬──────────────────────────────────────────────────────────┬──────────────────────────┤
   │                (1)    ┌─*CURLIB/───────┐                 │
   └─PPSRCFILE(────────────┼────────────────┼──file-name──)───┘
                           └─library-name/──┘
```

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.

```
├──┬─────────────────────────────────────┬───────────────────────────────────────┤
   │           (1)     ┌─*MODULE───┐      │
   └─PPSRCMBR(─────────┴─membername─┴──)─┘
```

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.

```
├──┬─────────────────────────────────────┬──────────────────────────────────────────┤
   │            (1)                      │
   └─PPSRCSTMF(──────────┬──────────┬──)─┘
                         ├─pathname─┤
                         └─*SRCSTMF─┘
```

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.

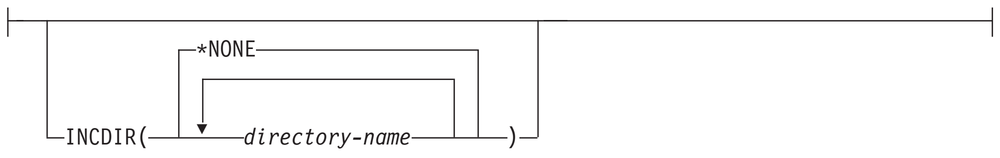
***SRCSTMF**
: If this setting is chosen, you must also select the SRCSTMF command option. Preprocessor output is saved to the current directory under the same base filename specified by the SRCSTMF command option, but with a filename extension of **.i**.

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.



***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.

```
├──┬─────────────────────────────────────────────────────┬──────────────┤
   │          ┌─*NONE────────────────────────────────┐   │
   └─CSOPT(───┼──────────────────────────────────────┼─)─┘
              └─'──compiler-service-options-string──'─┘
```

***NONE**
: Default setting. No compiler service options selected.

compiler-servicing-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.

```
├──┬──────────────────────────────────────────────────────────┬─────────┤
   │          ┌─*NONE──────────────────────────────────────┐   │
   └─LICOPT(──┼────────────────────────────────────────────┼─)─┘
              └─'──Licensed-Internal-Code-Options-String──'─┘
```

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 the *ILE Concepts Book* for more information about LICOPT options.

DFTCHAR

Instructs the compiler to treat all variables of type **char** as either signed or unsigned.

```
├──┬─────────────────────────────────┬──────────────────────────────────────────┤
   │               ┌─*UNSIGNED─┐     │
   └─DFTCHAR(──────┴─*SIGNED───┴──)──┘
```

***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.

```
├──┬─────────────────────────────────────────────────────┬──────────────────────┤
   │              ┌─*SOURCE──────────────────────────┐     │
   └─TGTCCSID(────┼─*JOB─────────────────────────────┼─)───┘
                  ├─*HEX─────────────────────────────┤
                  └─coded-character-set-identifier───┘
```

***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.

TEMPLATE

C++

Specifies options to customize C++ template generation.

```
├──┬─────────────────────────────────────────────────┬───────────────────────┤
   │            (1) (2) (3)    ┌─*NONE─────────────┐ │
   └─TEMPLATE(─────────────────┴┤ TEMPLATE Details ├┴─)─┘
```

TEMPLATE Details:

```
   ┌─*TEMPINC───────────┐   ┌─1───────┐   ┌─*NO────┐
├──┴─directory-pathname─┴───┼─────────┼───┼────────┼───────────────────────────┤
                            └─1 65535─┘   ├─*WARN──┤
                                          └─*ERROR─┘
```

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.

```
├─┬─────────────────────────────────────────────────────┬──────────────────────────────────────────┤
  │            (1) (2) (3)    ┌─*NONE──────────────┐    │
  └─TMPLREG(──────────────────┼─*DFT───────────────┼──)─┘
                              └─'──path-name──'────┘
```

Notes:

1 C++ compiler only

2 Create Module command only

3 Applicable only when using the Integrated File System (IFS)

The possible options are:

<u>*NONE</u>
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 or not 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.

```
├──┬──────────────────────────────────────┬─────────────────────────────────────┤
   │              (1) (2)     ┌─*YES─┐     │
   └─WEAKTMPL(────────────────┴─*NO──┴──)─┘
```

Notes:

1 C++ compiler only

2 Applicable only when using the Integrated File System (IFS)

The possible options are:

<u>*YES</u>
: 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.

Chapter 5. Using the ixlc Command to Invoke the C/C++ Compiler

The **ixlc** command lets you invoke the compiler and specify compiler options from a Windows client or iSeries Qshell command line. Module binder commands can be specified. The ixlc command can be used together with AIX or Windows make files to control compilation.

Using ixlc on a Windows Client

When using the Windows client version of **ixlc**, you can:

- Compile source files residing on your Windows client without first having to transfer them to an iSeries system. Header files, however, must reside on the iSeries system.
- Compile data management source code residing on an iSeries system.
- Compile IFS source code residing on an iSeries system.

You must install the CODE component of IBM WebSphere Development Studio Client for iSeries if you want to use **ixlc** from the command line of a Windows client.

The **ixlc** and **ixlclink** commands require you to sign on to the iSeries system. If you set a default host together with a valid user ID and password in the Communication Daemon, you will not have to sign on every time you want to use these commands.

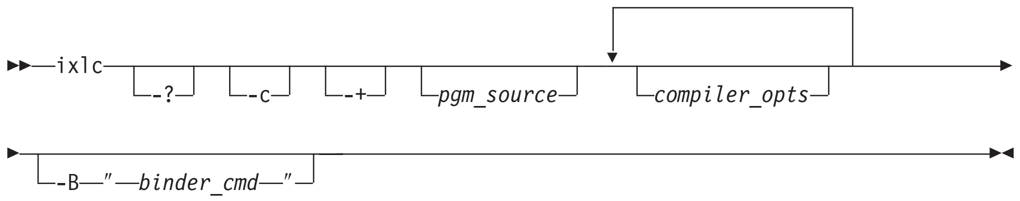
Using ixlc in Qshell

When using the iSeries version of **ixlc** on the ″green screen″ Qshell command line, you can:

- Compile data management source code residing on an iSeries system.
- Compile IFS source code residing on an iSeries system.
- Use header files residing on the iSeries system.

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.

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.

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
"MODULE" on page 86, "PGM" on page 86	`[*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.	

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
"SRCFILE" on page 87	`[*LIBL/ | *CURLIB/ | libraryname/] filename`	`-qsrcfile=[*LIBL/ | *CURLIB/ | libraryname/] filename`
"SRCMBR" on page 88	`*MODULE | mbrname`	`-qsrcmbr=mbrname`
"SRCSTMF" on page 89	*pathname*	(*none, uses default pathname*)
"TEXT" on page 90	`*SRCMBRTEXT | *BLANK | text`	`-qtext="text"`
"OUTPUT" on page 91	`*NONE`	`-qnoprint`
	`*PRINT`	`-qprint`
	filename	`-qoutput="filename"`

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
"OPTION" on page 92	*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

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
"CHECKOUT" on page 98	*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 101	10 | 20 | 30 | 40	-qoptimize=10 -qoptimize=20 -qoptimize=30 -qoptimize=40 -O
		-O is equivalent to specifying -qoptimize=40

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
"INLINE" on page 102	*OFF	-qnoinline
	*ON *NOAUTO | *AUTO 250 | 1-65535 | *NOLIMIT 2000 | 1-65535 | *NOLIMIT *NO | *YES	-qinline="*opt1 opt2 opt3 opt4*" where: • *opt1* is one of: – noauto – auto • *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 104	*KEEPILDATA | *NOKEEPILDATA	-qildta -qnoildta
"DBGVIEW" on page 105	*NONE | *ALL | *STMT | *SOURCE | *LIST	-qdbgview=none -qdbgview=all -qdbgview=stmt -qdbgview=source -qdbgview=list -g
		-g is equivalent to specifying -qdbgview=all
"DEFINE" on page 106	*NONE | *name* | *name=value*	-D*name*
		Defines *name* with a value of 1.
"LANGLVL" on page 107	*EXTENDED | *ANSI | *LEGACY	-qlanglvl=extended -qlanglvl=ansi -qlanglvl=compat366
"ALIAS" on page 108	*ANSI | *NOANSI | *ADDRTAKEN | *NOADDRTAKEN | *ALLPTRS | *NOALLPTRS | *TYPEPTR | *NOTYPEPTR	-qalias=ansi -qalias=noansi -qalias=addrtaken -qalias=noaddrtaken -qalias=allptrs -qalias=noallptrs -qalias=typeptr -qalias=notypeptr

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
"SYSIFCOPT" on page 109	*NOIFSIO | **IFSIO | *IFS64IO	-qnoifsio -qifsio -qifsio=64
	*ASYNCSIGNAL | *NOASYNCSIGNAL	-qasyncsignal -qnoasyncsignal
"LOCALETYPE" on page 110	*LOCALE | *LOCALEUCS2 | *LOCALEUTF | *CLD	-qlocale=locale -qlocale=localeucs2 -qlocale=localeutf -qlocale=cld
"FLAG" on page 111	0 | 10 | 20 | 30	-qflag=0 -qflag=10 -qflag=20 -qflag=30
"MSGLMT" on page 112	*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 113	*YES | *NO	-qreplace -qnoreplace
"USRPRF" on page 114	*USER | *OWNER	-quser -qowner
"AUT" on page 115	*LIBCRTAUT | *CHANGE | *USE | *ALL | *EXCLUDE	-qaut=libcrtaut -qaut=change -qaut=use -qaut=all -qaut=exclude
"TGTRLS" on page 116	*CURRENT | *PRV | *release_lvl*	-qtgtrls=*current -qtgtrls=*prv -qtgtrls=*VxRxMx*
"ENBPFRCOL" on page 118	*PEP	-qenbpfrcol=pep
	*ENTRYEXIT *NONLEAF	-qenbpfrcol=entryexitnonleaf
	*ENTRYEXIT *ALLPRC	-qenbpfrcol=entryexitallprc
	*FULL *NONLEAF	-qenbpfrcol=fullnonleaf
	*FULL *ALLPRC	-qenbpfrcol=fullallprc
"PFROPT" on page 119	*SETFPCA | *NOSETFPCA	-qsetfpca -qnosetfpca
	*NOSTRDONLY | *STRDONLY	-qnoro -qro
"PRFDTA" on page 120	*NOCOL | *COL	-qnoprofile -qprofile
		-qprfdta=*NOCOL -qprfdta=*COL
"TERASPACE" on page 121	*NO	-qteraspace=no
	*YES *NOTSIFC	-qteraspace=notsifc
	*YES *TSIFC	-qteraspace=tsifc

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
“STGMDL” on page 124	*SNGLVL | *TERASPACE | *INHERIT	-qstoragemodel=snglvl -qstoragemodel=teraspace -qstoragemodel=inherit
“DTAMDL” on page 125	*P128 | *LLP64	-qdatamodel=P128 -qdatamodel=LLP64
“RTBND” on page 126	*DEFAULT | *LLP64	-qrtbnd -qrtbnd=llp64
“PACKSTRUCT” on page 127	1 | 2 | 4 | 8 | 16 | *NATURAL	-qalign=1 -qalign=2 -qalign=4 -qalign=8 -qalign=16 -qalign=natural
“ENUM” on page 128	1 | 2 | 4 | *INT | *SMALL	-qenum=1 -qenum=2 -qenum=4 -qenum=int -qenum=small
“MAKEDEP” on page 129	*NODEP | *filename*	-M*makefile*
“PPGENOPT” on page 130	*NONE | *DFT	-P
	*RMVCOMMENT | *NORMVCOMMENT	-qppcomment -qnoppcomment
	*GENLINE | *NOGENLINE	-qppline -qnoppline
“PPSRCFILE” on page 131	*CURLIB/*filename*	-qppsrcfile=*CURLIB/*filename*
	libraryname/*filename*	-qppsrcfile=*libraryname*/*filename*
	filename	-qppsrcfile=*filename*
“PPSRCMBR” on page 132	*MODULE | *mbrname*	-qppsrcmbr=*module -qppsrcmbr=*mbrname*
“PPSRCSTMF” on page 133	*pathname* | *SRCSTMF	-qppfile=*filename* -qppfile=*srcstmf
“INCDIR” on page 134	*NONE | *pathname*	-I*pathname*
	When used on the command line, specifies directories on an iSeries system. Include environment variables are overwritten.	
“CSOPT” on page 135	*string*	-qcsopt=*string*
“LICOPT” on page 136	*NONE | *string*	-qlicopt=*string*
“DFTCHAR” on page 137	*SIGNED | *UNSIGNED	-qchar=signed -qchar=unsigned
“TGTCCSID” on page 138	*SOURCE | *JOB | *HEX | *ccsid#*	-qtgtccsid=source -qtgtccsid=job -qtgtccsid=hex -qtgtccsid=*ccsid#*
“TEMPLATE” on page 139	*NONE | *pathname*	-qnotempinc -qtempinc=*pathname*
	1 - 65535	-qtempmax=*1-65535*
	*NO | *WARN | *ERROR	-qtmplparse=no -qtmplparse=warn -qtmplparse=error

Create Module/Create Bound Program Options	Option Settings	ixlc Equivalents and Notes
"TMPLREG" on page 141	`*DFT` | `*NONE`	`-qtmplreg` `-qnotmplreg`
"WEAKTMPL" on page 142	`*YES` | `*NO`	`-qweaktmpl` `-qnoweaktmpl`

Chapter 6. Using ixlclink to Create Programs

The **ixlclink** command lets you invoke the OS/400 Create Program (CRTPGM) and Create Service Program (CRTSRVPGM) commands from a personal computer workstation. Command parameters are passed by **ixlclink** to the OS/400 CRTPGM or CRTSRVPGM programs, which in turn bind modules residing on an iSeries system into an ILE program or service program.

You must install the CODE component of IBM WebSphere Development Studio Client for iSeries if you want to use the **ixlclink** command from the command line of a Windows client.

Examples of ixlclink Usage

1. The following ixlclink command invokes the OS/400 CRTPGM command to create a program.

   ```
   c:\>ixlclink -qpgm=usr/simple -qgen -qnodupproc -qmodule=usr/simplec
        "-qtext='simple c program' "
   ```

   It is equivalent to issuing the following CL command:

   ```
   CRTPGM PGM(usr/simple) module(usr/simplec) text('simple c program' )
          option( *gen *nodupproc)
   ```
2. The following ixlclink command invokes the OS/400 CRTSRVPGM command to create a service program.

   ```
   c:\>ixlclink -qsrvpgm=usr/simple "-qbnddir=temp/a temp/c" -qgen
      -qnodupproc -qmodule=usr/simplec "-qtext='simple service program' "
   ```

   is equivalent to issuing the following CL command:

   ```
   CRTSRVPGM SRVPGM(usr/simple) bnddir(temp/a temp/b) module(usr/simplec)
        text('simple service program' )  option( *gen *nodupproc)
   ```
3. If the command line parameter includes one or more spaces, you need to use ″ ″ to enclose the parameter. For example:

   ```
   "-qbnddir=temp/a  temp/b"
   ```

   is seen by CL as:

   ```
   bnddir(temp/a temp/b)
   ```
4. If you want to use * as part of a string, you need to use ′ ′ to enclose the string. For example:

   ```
   -qtext='*blank'
   ```

   is seen by CL as:

   ```
   -qtext=*blank
   ```
5. If there is space in a string associated with a command line parameter, you need to use ′ ′ to quote the string and ″ ″ to quote the command line. For example:

   ```
   "-qtext='simple c'"
   ```

   is seen by CL as:

```
text('simple c')
```

For more information on creating programs or service programs, see *Creating a Program* and *Creating a Service Program* in *WebSphere Development Studio: ILE C/C++ Programmer's Guide*.

ixlclink Command Options

Most CRTPGM and CRTSRVPGM command options have ixlclink counterparts, as listed in the table below:

CRTPGM and CRTSRVPGM Command Options	Option Settings	ixlclink Equivalents
(none)	*(none)*	-?
		Displays help for the ixlclink command.
PGM	[*CURLIB/ | *libraryname/*]*name*	-qpgm=[*CURLIB/ | *libraryname/*]*name*
SRVPGM	[*CURLIB/ | *libraryname/*]*name*	-qsrvpgm=[*CURLIB/ | *libraryname/*]*name*
MODULE	[*LIBL/ | *CURLIB/ | *USRLIBL/ | *libraryname/*] *PGM | *SRVPGM |*ALL | *name*	-qmodule=[*LIBL/ | *CURLIB/ | *USRLIBL/ | *libraryname/*]*PGM | *SRVPGM *ALL | *name*
TEXT	*ENTMODTEXT | *BLANK | *text*	-qtext="*text*"
ENTMOD	[*LIBL/ | *CURLIB/ | *USRLIBL/ | *libraryname/*] *FIRST | *ONLY |*PGM | *modulename*	-qentmod=[*LIBL/ | *CURLIB/ | *USRLIBL/ | *libraryname/*]*FIRST | *ONLY | *PGM | *modulename*
BNDSRVPGM	[*LIBL/ | *libraryname/*]*NONE | *ALL | *srv_pgmname*	-qbndsrvpgm==[*LIBL/ | *libraryname/*]*NONE | *ALL | *srv_pgmname*
BNDDIR	[*LIBL/ | *CURLIB/ | *USRLIBL/ | *libraryname/*] *NONE | *dir*	-qbnddir= =[*LIBL/ | *CURLIB/ | *USRLIBL/ | *libraryname/*]*NONE | *dir*
ACTGRP	*NEW | *CALLER | actgrpname	-qactgrp=*NEW -qactgrp=*CALLER -qactgrp=actgrpname
OPTION	*GEN | *NOGEN	-qgen -qnogen
	*DUPPROC | *NODUPPROC	-qdupproc -qnodupproc
	*DUPVAR | *NODUPVAR	-qdupvar -qnodupvar
	*WARN | *NOWARN	-qwarn -qnowarn
	*RSLVREF | *NORSLVREF	-qrslvref -qnorslvref
DETAIL	*NONE | *BASIC | *EXTENDED | *FULL	-qdetail=*NONE -qdetail=*BASIC -qdetail=*EXTENDED -qdetail=*FULL
ALWUPD	*YES | *NO	-qalwupd -qnoalwupd
ALWLIBUPD	*YES | *NO	-qalwlibupd -qnoalwlibupd

CRTPGM and CRTSRVPGM Command Options	Option Settings	ixlclink Equivalents
REPLACE	*YES | *NO	-qreplace -qnoreplace
USRPRF	*USER | *OWNER	-qusrprf=*USER -qusrprf=*OWNER
AUT	*LIBCRTAUT | *CHANGE | *USE | *ALL | *EXCLUDE	-qaut=*libcrtaut -qaut=*all -qaut=*change -qaut=*use -qaut=*exclude
TGTRLS	*CURRENT | *PRV | *release_lvl*	-qtgtrls=*current -qtgtrls=*prv -qtgtrls=V*x*R*x*M*x*
ALWRINZ	*YES | *NO	-qalwrinz -qnoalwrinz
EXPORT	*SRCFILE | *ALL	-qexport=*srcfile -qexport=*all
SRCFILE	[*LIBL/ | *CURLIB/ | *libraryname*/] QSRVSRC | *filename*	-qsrcfile=[*LIBL/ | *CURLIB/ | *libraryname*/] QSRVSRC | *filename*
SRCMBR	*SRVPGM | membername	-qsrcmbr=*membername* -qsrcmbr=*SRVPGM

For more information about CRTPGM or CRTSRVPGM programs and syntax, see *CRTPGM (Create Program) Command Description* and *CRTSRVPGM (Create Service Program) Command Description*, available on the Web from the *iSeries 400 Information Center* at:

```
http://www.ibm.com/eserver/iseries/infocenter
```

After selecting your geographic location along with language and operating system level, select `Programming -> CL -> Alphabetical List of Commands` from the contents menu. Alternately, use the search function to search for the terms `CRTPGM` and `CRTSRVPGM`.

Chapter 7. I/O Considerations

This chapter provides information on:

- Data Management Operations on Record Files
- Data Management Operations on Stream Files
- C Streams and File Types
- DDS-to-C/C++ Data Type Mappings

Data Management Operations on Record Files

For more information about data management operations and ILE C/C++ functions available for record files, see the *File Systems and Management* section in the *Database and File Systems* category at the iSeries 400 Information Center Web site:

```
http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm
```

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 *File Systems and Management* section in the *Database and File Systems* category at the iSeries 400 Information Center Web site:

```
http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm
```

C Streams and File Types

The following table summarizes which file types are supported as streams.

Table 2. 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 3. 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)
Z - time stamp	26	none	char field[26];

Note: 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*, available in PDF and HTML formats from the iSeries 400 Information Center Web site at:

```
http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm
```

Appendix. 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 4. 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
IGS (interchange group separator)	0x1d
IRS (interchange record separator)	0x1e
IUS/ITB (interchange unit separator or intermediate transmission block)	0x1f
DS (digit select)	0x20

Table 4. Internal Hexadecimal Representation (continued)

Print representation	**Internal representation**
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

Bibliography

For additional information about topics related to ILE C/C++ programming, refer to the following IBM publications:

- *CL Programming*, SC41-5721-05, provides a wide-ranging discussion of iSeries programming topics including 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 OS/400 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 OS/400 graphical data display manager (GDDM) to write graphics application programs. This manual provides detailed descriptions of all graphics routines available in GDDM. Also provides information about high-level language interfaces to GDDM.
- *WebSphere Development Studio: ILE C/C++ Programmer's Guide*, SC09-2712-05, 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.
- *WebSphere Development Studio: C/C++ Language Reference*, SC09-4815-00, 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++ Run-Time Library Functions*, SC41-5607-01, provides reference information about C for AS/400 library functions, including Standard C library functions and C for AS/400 library extensions. Examples are provided and considerations for programming are also discussed.
- *ILE Concepts*, SC41-5606-06, explains concepts and terminology pertaining to the Integrated Language Environment architecture of the OS/400 licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- *System API Programming*, SC41-5800-00, provides information for the experienced application and system programmers who want to use the OS/400 application programming interfaces (APIs). Provides getting started and examples to help the programmer use APIs.

Notices

Programming Interface Information

This book is intended to help you create Integrated Language Environment C and C++ programs. It contains information necessary to use the Integrated Language Environment C/C++ compiler and documents general-use programming interfaces and associated guidance information provided by the Integrated Language Environment C/C++ compiler.

Trademarks and Service Marks

Industry Standards

The Integrated Language Environment C/C++ compiler and run-time library are designed according to the ANSI for C Programming Languages - C ANSI/ISO 9899-1990 standard, and the November 1997 ANSI C++ Draft Standard.

Index

Program Number: 5722–WDS

SC09-4816-03