VisualAge® C++ Professional for AIX®

Batch Compiler and Other Tools

Version 5.0

Note!

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

Edition Notice

This edition applies to Version 5.0 of IBM® VisualAge C++ and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

Notices

Note to U.S. Government Users Restricted Rights -- use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interface allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

Industry Standards

VisualAge C++ Professional for AIX, Version 5.0 supports the following standards:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The ISO/IEC 9945–1:1990/IEEE POSIX 1003.-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

About This Book

The information in this PDF document is also available in the online help.

To find this information, or any topics listed in this document as Related Concepts, Related Tasks, or Related References, simply type the topic title into the search bar in the top frame of your browser in the online help.

For some topics, the suggested references may already be contained in this document. In such cases, there is a cross-reference to the page on which the related topic appears.

Chapter 1. Batch Compiling and Linking

VisualAge C++ Batch Compilers

You can use IBM VisualAge C++ in a batch mode as a C compiler for files with a .c (small c) suffix, or as a C++ compiler for files with a .C (capital C), .cc, .cpp, or .cxx suffix. The compiler processes your text-based program source files to create an executable object module.

Note: Use of the xlC Command in this Information
Throughout these information panels, the **xlC** command is used to describe the actions of the compiler. In most cases, you should use the **xlC** command to compile your C or C++ source files.

The **xlC_r** and **xlC128** commands specify additional libraries, macros, or options that are not automatically included or set by the **xlC** command. Besides these differences, these commands may be considered functionally equivalent, so that any mention of one implies the other. This is also true for the **xlc, cc**, **cc_r** and **cc128** commands.

RELATED CONCEPTS

Compiler Modes
Batch Compiler Options
Types of Input Files
Types of Output Files
Compiler Messages and Listings
Incremental C++ Builds

RELATED TASKS

Invoke the Batch Compiler
Invoke the Linkage Editor
Migrate From Batch to Incremental

RELATED REFERENCES

List of Batch Compiler Options and Their Defaults
Equivalent Batch Compile-Link and Incremental Build Options

Compiler Modes

Several forms of VisualAge C++ compiler invocation commands support various version levels of the C and C++ languages. Normally, you should use the **xlc** command to compile your C source files and the **xlC** command to compile your C++ source files. You can, however, use other forms of the command if your particular environment and file systems require it. The various compiler invocation commands are:

Batch Invocation Commands for IBM VisualAge C++

xlC	xlC128	xlC_r	xlC128_r	xlC_r4	xlC_r7	xlC128_r7
xlc	xlc128	xlc_r	xlc128_r	xlc_r4	xlc_r7	xlc128_r7

cc	cc128	cc_r	cc_r4	cc_r7	cc128_r7
CC_4	**CC_r4**				
c89					

The four basic compiler invocation commands appear as the first entry of each line in the table above. Select a basic invocation using the following criteria:

xlC	Invokes the compiler so that source files are compiled as C++ language source code. Files with **.c** suffixes, assuming you have not used the "+" on page 89 compiler option, are compiled as C language source code with a default language level of **ansi**, and compiler option "ansialias" on page 95 to allow type-based aliasing. If any of your source files are C++, you must use this invocation to link with the correct runtime libraries.
xlc	Invokes the compiler for C source files with a default language level of **ansi**, and compiler option "ansialias" on page 95 to allow type-based aliasing.
cc	Invokes the compiler for C or C++ source files with a default language level of **extended** and compiler options "ro" on page 210 and "roconst" on page 210 (to provide compatibility with the RT compiler and placement of string literals or constant values in read/write storage). Use this invocation for legacy C code that does not require compliance withANSI C.
c89	Invokes the compiler for C or C++ source files, with a default language level of **ansi**, and specifies compiler options "ansialias" on page 95 (to allow type based aliasing) and "longlong" on page 176 (disabling use of **long long**), and sets "D" on page 112_ANSI_C_SOURCE (for ANSI-conformant headers). Use this invocation for strict conformance to the ANSI standard (ISO/IEC 9899:1990).

IBM VisualAge C++ provides variations on the four basic batch compiler invocations. These variations are described below:

128-suffixed Invocations	All **128**-suffixed invocation commands are functionally similar to their corresponding base compiler invocations. They specify the "ldbl128, longdouble" on page 173 option, which increases the length of **long double** types in your program from 64 to 128 bits. They also link with the 128 versions of the C and C++ runtimes.
_r-suffixed Invocations	All **_r**-suffixed invocations additionally set the macro names **"D" on page 112_THREAD_SAFE** and add the libraries **"L" on page 159/usr/lib/threads**, **"l" on page 160c_r** and **-lpthreads**. The compiler option -qthreaded is also added. Use these commands if you want to create Posix threaded applications. AIX 4.1 and 4.2 support Posix Draft 7. AIX 4.3 supports Draft 10. The _r7 invocations are provided on AIX 4.3 to help with migration to Draft 10. See "threaded" on page 229 for additional information. The _r4 invocations should be used for DCE threaded applications.

Migrating AIX Version 3.2.5 DCE Applications to AIX Version 4.1
The main invocation commands (except **c89**) have additional **_r4**-suffixed forms. These forms provide compatibility between DCE applications written for AIX Version 3.2.5 and AIX Version 4. They link your application to the correct AIX Version 4 DCE libraries, providing compatibility between the latest version of the pthreads library and the earlier versions supported on AIX Version 3.2.5.

RELATED TASKS

"Invoke the Batch Compiler" on page 15

RELATED REFERENCES

/etc/vac.cfg - Default Batch Configuration File

Batch Compiler Options

Compiler options perform a wide variety of functions, such as setting compiler characteristics, describing the object code and compiler output to be produced, and performing some preprocessor functions. You can specify compiler options in one or more of three ways:

- on the command line
- in a batch configuration file (.cfg)
- in your source program

When specifying compiler options in more than one of the above locations, it is possible for option conflicts and incompatibilities to occur. IBM VisualAge C++ resolves these conflicts and incompatibilities in a consistent fashion.

RELATED TASKS

"Invoke the Batch Compiler" on page 15

RELATED REFERENCES

"Resolving Conflicting Compiler Options" on page 43
"List of Batch Compiler Options and Their Defaults" on page 24

Types of Input Files

You can input the following types of files to the VisualAge C++ batch compilers:

C and C++ Source Files
These are files containing a C or C++ source module.

To use the compiler as a C language compiler to compile a C language source file, the source file must have a .c (lowercase c) suffix, for example, mysource.c.

To use the compiler as a C++ language compiler, the source file must have a .C (uppercase C), .cc, .cpp, or .cxx suffix. To compile other files as C++ source files, use the **-+** compiler option. All files specified with this option with a suffix other than .o, .a, or.s, are compiled as C++ source files.

The compiler will also accept source files with the .i suffix. This extension designates preprocessed source files.

The compiler processes the source files in the order in which they appear. If the compiler cannot find a specified source file, it produces an error message and the compiler proceeds to the next specified file. However, the link editor will not be run and temporary object files will be removed.

Your program can consist of several source files. All of these source files can be compiled at once using only one invocation of **xlC**. Although more than one source file can be compiled using a single invocation of the compiler, you can specify only one set of compiler options on the command line per invocation. Each distinct set of command-line compiler options that you want to specify requires a separate invocation.

By default, the **xlC** command preprocesses and compiles all the specified source files. Although you will usually want to use this default, you can use the **xlC** command to preprocess the source file without compiling by specifying either the "E" on page 116 or the "P" on page 195 option. If you specify the "P" on page 195 option, a preprocessed source file, *file_name*.i, is created and processing ends.

The "E" on page 116 option preprocesses the source file, writes to standard output, and halts processing without generating an output file.

Preprocessed Source Files
Preprocessed source files have a .i suffix, for example, file_name.i.

The **xlC** command sends the preprocessed source file, *file_name*.i, to the compiler where it is preprocessed again in the same way as a .c file. Preprocessed files are useful for checking macros and preprocessor directives.

Object Files
Object files must have a .o suffix, for example, year.o.

Object files, library files, and nonstripped executable files serve as input to the linkage editor. After compilation, the linkage editor links all of the specified object files to create an executable file.

Assembler Files
Assembler files must have a .s suffix, for example, check.s.

Assembler files are assembled to create an object file.

Nonstripped Executable Files	Extended Common Object File Format (XCOFF) files that have not been stripped with the AIX **strip** command can be used as input to the compiler. See the **strip** command in the *AIX Version 4 Commands Reference*, and the description of a.out file format in the *AIX Version 4 Files Reference* for more information.

RELATED CONCEPTS

"Types of Output Files"

Types of Output Files

You can specify the following types of output files when invoking IBM VisualAge C++ batch compilers.

Executable File	By default, executable files are named a.out. To name the executable file something else, use the "o" on page 190*file_name* option with the invocation command. This option creates an executable file with the name you specify as *file_name*. The name you specify can be a relative or absolute path name for the executable file. The format of the a.out file is described in the *AIX Version 4 Files Reference*.
Object Files	Object files must have a .o suffix, for example, year.o, unless the **-o***filename* option is specified. If you specify the "c" on page 103 option, an output object file, *file_name*.o, is produced for each input source file *file_name*.c. The linkage editor is not invoked, and the object files are placed in your current directory. All processing stops at the completion of the compilation. You can link-edit the object files later into a single executable file using the **xlC** command.
Assembler Files	Assembler files must have a .s suffix, for example, check.s. They are created by specifying the "S" on page 213 option. Assembler files are assembled to create an object file.
Preprocessed Source Files	Preprocessed source files have a .isuffix, for example, tax_calc.i. To make a preprocessed source file, specify the "P" on page 195 option. The source files are preprocessed but not compiled. You can also use redirect the output from the "E" on page 116 option to generate a preprocessed file that contains #line directives. A preprocessed source file, *file_name*.i, is produced for each source file, *file_name*.c.
Listing Files	Listing files have a .lst suffix, for example, form.lst. Specifying any one of the listing-related options to the invocation command produces a compiler listing (unless you have specified the "noprint" on page 189 option). The file containing this listing is placed in your current directory and has the same file name (with a .lst extension) as the source file from which it was produced.
Target File	Output files associated with the "M" on page 177 or "makedep" on page 182 options have a .u suffix, for example, conversion.u. The file contains targets suitable for inclusion in a description file for the AIX **make** command. A .u file is created for every input C or C++ file. .u files are not created for any other files (unless you use the **-+** option so other file suffixes are treated as .C files).

RELATED CONCEPTS

"Types of Input Files" on page 4

Compiler Message and Listing Information

When the compiler encounters a programming error while compiling a C or C++ source program, it issues a diagnostic message to the standard error device and to the listing file.

The compiler issues messages specific to the C or C++ language, and XL messages common to all XL compilers.

For details about options that affect the information available in compiler listings, see "Options that Specify Compiler Output" on page 40

▶ C

If you specify the compiler option "srcmsg (C Only)" on page 219 and the error is applicable to a particular line of code, the reconstructed source line or partial source line is included with the error message in the stderr file. A reconstructed source line is a preprocessed source line that has all the macros expanded.

If the error is identifiable within the source line, a *finger line* under the source line points to the column position of the error. For example:

```
          10 | int add(int, int)
               ....a...b....c...
     a - 1506-166 (S) Definition of function add requires parentheses.
     b - 1506-172 (S) Parameter type list for function add contains
     parameters without identifiers.
     c - 1506-172 (S) Parameter type list for function add contains
     parameters without identifiers.
```

The compiler also places messages in the source listing if you specify the "source" on page 217 option.

If "langlvl" on page 160 is set to **ansi**, compile-time messages about incorrect **#pragma** directives are not generated.

You can control the diagnostic messages issued, according to their severity, using either the "flag" on page 126 option or the "w" on page 240 option. To get additional informational messages about potential problems in your program, use the "info" on page 144 option.

Compiler Listings

The listings produced by the compiler are a useful debugging aid. By specifying appropriate options, you can request information on all aspects of a compilation. The listing consists of a combination of the following sections:

- Header section that lists the compiler name, version, and release, as well as the source file name and the date and time of the compilation
- Source section that lists the input source code with line numbers. If there is an error at a line, the associated error message appears after the source line and a pointer to the error location.
- Options section that lists the options that were in effect during the compilation
- Attribute and cross-reference listing section that provides information about the variables used in the compilation unit

- File table section that shows the file number and file name for each main source file and include file
- Compilation epilogue section that summarizes the diagnostic messages, lists the number of source lines read, and indicates whether the compilation was successful
- Object section that lists the object code

Each section, except the header section, has a section heading that identifies it. The section heading is enclosed by angle brackets:

RELATED REFERENCES

"Options that Specify Compiler Output" on page 40
"Batch Compiler Message Format" on page 46
"Message Severity Levels and Compiler Response" on page 44

Set Up the Batch Compilation Environment

Before you compile your C or C++ programs, you must set up the environment variables and the configuration file for your application.

Set Environment Variables to Select 64- or 32-bit Modes (AIX 4.3 Only)
You can set the OBJECT_MODE environment variable to specify a default compilation mode. Permissible values for the OBJECT_MODE environment variable are:

(unset)	Compiler programs generate and/or use 32-bit objects.
32	Compiler programs generate and/or use 32-bit objects.
64	Compiler programs generate and/or use 64-bit objects.
32_64	Set the compiler programs to accept both 32- and 64-bit objects. The compiler never functions in this mode, and using this choice may generate an error message, depending on other compilation options set at compile-time.

See "Specify Batch Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 21 for more information.

Set Parallel Processing Run-time Options (C Only)

The XLSMPOPTS environment variable sets options for programs using loop parallelization. For example, to have a program run-time create 4 threads and use dynamic scheduling with chunk size of 5, you would set the XLSMPOPTS environment variable as shown below:

```
XLSMPOPTS=PARTHDS=4:SCHEDULE=DYNAMIC=5
```

Additional environment variables set options for program parallelization using OpenMP-compliant directives.

See “IBM Run-time Options for Parallel Processing (C Only)” on page 63 and “OpenMP Run-time Options for Parallel Processing (C Only)” on page 66 for more information.

Set Environment Variables for the Message and Help Files
Before using the compiler, you must install the message catalogs and help files and set the following two environment variables:

LANG	Specifies the national language for message and help files.
NLSPATH	Specifies the path name of the message and help files.

The **LANG** environment variable can be set to any of the locales provided on the system. See the description of locales in *AIX General Programming Concepts for IBM RISC System/6000*® for more information.

The national language code for United States English is **en_US**. If the appropriate message catalogs have been installed on your system, any other valid national language code can be substituted for **en_US**.

To determine the current setting of the national language on your system, use the both of the following **echo** commands:

```
echo $LANG
echo $NLSPATH
```

The **LANG** and **NLSPATH** environment variables are initialized when the operating system is installed, and might differ from the ones you want to use.

You use different commands to set the environment variables depending on whether you are using the Bourne shell (**bsh**or **sh**), Korn shell (**ksh**), or C shell (**csh**). To determine the current shell, use the **echo** command:

```
echo $SHELL
```

The Bourne-shell path is **/bin/bsh** or **/bin/sh**. The Korn shell path is **/bin/ksh**. The C-shell path is **/bin/csh**.

For more information about the **NLSPATH** and **LANG** environment variables, see *AIX Version 4 System User’s Guide: Operating System and Devices*. The AIX international language facilities are described in the *AIX General Programming Concepts for IBM RISC System/6000.*

Set Environment Variables in bsh, ksh, or sh Shells
To set the environment variables from the Bourne shell or Korn shell, use the following commands:

```
LANG=en_US
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/L/%N
export LANG NLSPATH
```

To set the variables so that all users have access to them, add the commands to the file **/etc/profile**. To set them for a specific user only, add the commands to the file **.profile** in the user’s home directory. The environment variables are set each time the user logs in.

Set Environment Variables in csh Shell

To set the environment variables from the C shell, use the following commands:

```
setenv LANG en_US
setenv NLSPATH /usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/L/%N
```

In the C shell, you cannot set the environment variables so that all users have access to them. To set them for a specific user only, add the commands to the file **.cshrc** in the user's home directory. The environment variables are set each time the user logs in.

RELATED TASKS

"Specify Batch Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 21

RELATED REFERENCES

"IBM Run-time Options for Parallel Processing (C Only)" on page 63
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66

Structure a Program that Uses Templates (Batch Compiler Only)

The following class template, `Stack`, is used as an example in the sections that follow. `Stack` implements a stack of items. The overloaded operators << and >> are used to push items to the stack and pop items from the stack. Both return an integer result: 1 = success, 0 = failure. The declaration of the `Stack` class template is contained in the file **stack.h**

Declaration of Stack in stack.h

```
typedef enum{tr,fl} Bool;
template <class Item, int size> class Stack {
public:
int operator << (Item item);  // Push operator
int operator >> (Item& item); // Pop operator
Stack(Bool p=fl) {top = 0;}   // Constructor defined inline
private:
Item stack[size]; // The stack of elements
int top; // Index to top of stack
};
```

In this example, the constructor function is defined inline, and has external linkage. The other functions are defined using separate function templates. These function templates are contained in the file **stack.c**

Declaration of operator Functions in stack.c

```
template <class Item, int size>
int Stack<Item,size>::operator << (Item item) {
if (top >= size) return 0;
stack[top++] = item;
return 1;
}
template <class Item, int size>
int Stack<Item,size>::operator >> (Item& item)
{
if (top <= 0) return 0;
item = stack[–top];
return 1;
}
```

Template Functions Declared Inline and Template Functions With Internal Linkage

If a template function is considered to be *inline* if one of the following applies :

- it is defined within a class definition
- it is declared using the `inline` specifier

An inline function is defined in each translation unit in which it is used and has exactly the same definition in each case. Thus, the compiler generates the same function in each of the compilation units where the template function is instantiated. The compiler may also inline the function for you (inline substitution of the function body at the point of call, similar to macro substitution).

A namespace scope template function has internal linkage if it is explicitly declared static. No other template function has internal linkage. The `inline` function specifier does not affect the linkage of a template function. You must define a template function that has internal linkage within the compilation unit in which it is used (implicitly instantiated, explicitly instantiated or specialized) because a name that has internal linkage cannot be referred to by other names from other translation units.

The definition of a template function must be in scope (visible) at the point of an explicit specialization or instantiation. On the other hand, the only requirement for a template function to be implicitly instantiated, is that the function declaration has to be in scope at the point of instantiation.

In the `Stack` template class example, the constructor is defined inline in the class template declaration. As a result, any compilation unit that uses an instance of the `Stack` class will have the appropriate constructor generated as an inline function by the compiler.

Template Functions Defined within the Compilation Unit

If a compilation unit explicitly instantiates or specialize a template function with a set of arguments, the compiler generates the function definition. At link-edit time, all references to this function in all compilation units are resolved to this function definition. If different compilation units explicitly instantiate or specialize the same template function with the same set of arguments, the compiler reports an error.

If a compilation unit contains a template declaration that defines a function, the compiler generates the function code for all functions that are explicitly specialized or instantiated within the compilation unit and for which the template function definition is in scope at the point of instantiation

More than one compilation unit could meet this criteria. If so, several compilation units may generate function code for the same template function.

The link-edit step does not remove any unused template function code from the executable program. Therefore, if the same template code that defines functions is contained in multiple compilation units, you may generate a very large executable program. In the Stack class template example, for any compilation units that include the file **stack.c**, the compiler generates function code for each Stack class instance in that compilation unit. For example, a compilation unit that contains :

```
#include "stack.h"
#include "stack.c"
void Swap(int &i, Stack& s)
{
int j;
s >> j;
s << j;
i = j;
}
```

will automatically generate code for these functions :

```
Stack<int,20>::operator << (int)
Stack<int,20>::operator>> (int&)
```

Generate Template Functions Automatically

To avoid producing template code for the same function multiple times, use the compiler to automatically generate the template functions. This is the recommended way to use templates with the batch compiler.

The compiler can generate template function code automatically, provided the template functions are referenced but not defined in your program code. To use this method, you must generate a special file called a *template-implementation* file that the compiler uses to generate the function code.

With this template-implementation file, the compiler generates each function definition only once for the whole program. The compiler determines what instances of the function must be created and avoids generating multiple copies of the template functions.

You can specify more than one template implementation file for a header file using the #pragma implementation directive.

To generate template functions automatically:

1. Declare but do not define the template function.
2. Place the class or function template declaration in a header file and include this header file in your source program by using the #include directive. If the template function has class (template) scope, its declaration is part of the class (template) definition. If the template function has namespace scope, you must declare but not define the function using a function template.
3. Create a special template-implementation file for each of the header files that contain these template declarations. Use the same name for the template-implementation file as for the header file but use a .c (lower case c) instead of a .h suffix. Place these template-implementation files in the same directories as their corespondent .h files.
4. Define all the functions declared in the header file in this template-implementation file. These functions can be partial function specializations, pure template function definitions, or both. The compiler only picks up C++ code that defines partial function specialization or pure function templates, and some other file (included in your project) that eventually includes the template-include file actually requires their instantiation.
5. Place the definitions of any types (classes) that are used in template arguments in header files (so with other words the types used for implicit instantiation). If the class definitions require other header files, use the #include directive to include them in either your implementation file or in the .h file (not in the main file). If any type (class) is required to declare a template function (the types are used as parameter types), place them either in the template declaration file (the .h file) or in a separate header file but make sure to include it in the template declaration file using the #include directive and NOT directly in your main file (for example the Bool type in the stack example). Do not put the definitions of any classes that are used in template arguments (the implicit

instantiation) or in template function definition in your source file. This is because if a user-defined type is used in an implicit function instantiation, the compiler will automatically include the file in the tempinc generated template file, but won't do that for types used in template function definition or declaration. <.li>

6. If you compile and then link at a different time, repeat any compiler options that you have specified at compile time, when you link. Using the same compiler options allows the compiler to properly compile the template-include files that is generated at compile time. For example, use the same path names for the -Idirectory option so that the compiler uses the same include files.

Note: Do not use the -qnotempinc option. Automatic function generation is disabled by the -qnotempinc option. In the `Stack` class template example, the **stack.h** header file is included in any compilation units that use instances of the class. The **stack.c** file is not included by any of these compilation units. The compiler uses it to build the necessary functions. For example, a main compilation unit may contain:

```
   #include "stack.h"
void Swap(int &i, Stack& s)
{
int j;
s >> j;
s << j;
i = j;
}
```

During the link-edit step, the compiler will automatically generate code for these functions :

```
Stack<int,20>::operator << (int)
Stack<int,20>::operator>> (int&)
```

Note to Users of xlC Version 3.x: Previous versions of xlC instantiated every method of a class whether used or not. With VisualAge C++ V5.0, xlC only instantiates the methods that are used. This may result in unresolved symbols for poorly organized code. Check your organization. Reference symbols in the source file where the symbols were generated with xlC V3.x.

How the Compiler Generates the Function Definitions

During compilation of your program, the compiler builds up a special template instantiation file for each header file that contains template functions that require code generatation. The compiler stores these template instantiation files in the `tempinc` subdirectory of the working directory. It creates the tempinc subdirectory if one does not already exist.

Before link-editing your program, the compiler checks the contents of the tempinc subdirectory, compiles its template instantiation files, and generates the necessary template function code.

You can rename the template-implementation file or place it in a different directory with the #pragma implementation directive. If you designate a relative path name, the path must be relative to the directory containing the header file.

In the Stack class template example, if you want to use the file **stack.defs** as the template implementation file instead of **stack.c**, add the line `#pragma` `implementation("stack.defs")` anywhere in the **stack.h** file. The compiler expects to find the template implementation file **stack.defs** in the same directory as **stack.h**.

Specifying the Template-Implementation File
Define all the functions declared in the header file in the template-implementation files. These definitions can be explicit function instantiations (specializations), template definitions, or both. If you include these explicit specializations, the compiler uses them rather than those generated from the template when it processes the template instantiation file.

If you use a class template argument and the class definition is needed in the template-implementation file to generate the template function, make sure you include the class definition in the header file. The compiler includes the header file in the template instantiation file. This makes the class definition available when the function definition is compiled.

Specifying a Different Path for the tempinc Subdirectory
By default, the compiler builds and compiles the special template-include files in the **tempinc** subdirectory of the working directory. To redirect these files to another directory, use the -qtempinc option. If you specify a directory, make sure you specify it consistently for all compilations and link-edits of your program.

Regenerating the Template Instantiation File
The compiler builds a template instantiation file corresponding to each header file containing template function declarations. After the compiler creates one of these files, it may add information to it as each compilation unit is compiled. The compiler never removes information from the file.

As you develop your program, you may remove function instantiations or reorganize your program, so that the template instantiation files become obsolete. Since the compiler does not remove information from the template instantiation files, you may want to delete one or more of these files and recompile your program periodically. To regenerate all of the template instantiation files, delete the tempinc directory and recompile your program.

If a compiler-generated file in a tempinc directory has compiler errors, the file will be recompiled whenever a link is done using the **xlC** with that tempinc directory. To avoid this recompilation, either fix the errors in the file or remove it from the tempinc directory.

Breaking the Template Instantiation File into More Than One File
Normally the compiler generates one template instantiation file for each template implementation file (either the default one or one specified by #pragma implementation). When compiled, the template instantiation file may be too large to be created by the compiler. If so, you can break the template instantiation file into more than one file using the -qtempmax=number compiler option. More than one object file will be created, all of them smaller in size than the first one.

Contents of Template Instantiation File
This section contains two examples of template instantiation files. The first is an example showing the information that would be in a typical template instantiation file. The second is the file produced for Stack class template example.

Example of a Typical Template-Include File
The following example shows the layout of a typical template instantiation file

generated by the compiler. The compiler does not remove information from the file. You can edit these files but it is neither necessary nor advisable.

```
/*0698421265*/#include "/home/myapp/src/list.h" 1
/*0000000000*/#include "/home/myapp/src/list.c"     2
/*0698414046*/#include "/home/myapp/src/mytype.h"       3
/*0698414046*/#include "/usr/vacpp/include/iostream.h"     4
template int List<MyType>::foo();                       5
template ostream& operator<<(ostream&, List<MyType>);          6
template int count(List<MyType>); 1. 2. 3. 4.           7
```

Line 1. The header file that corresponds to the template-include file. The number in comments at the start of each #include line (in this case, /*0698421265*/) is a timestamp for the included file. The compiler uses this number to determine if the template-include file should be recompiled. A time stamp of zeroes (as in line 2.) means the compiler is to ignore the timestamp.
Line 2. The template implementation file that corresponds to the header file in line 1.
Line 3. Another header file that the compiler needs to compile the template-include file. All other header files that the compiler needs to compile the template-include file are inserted at this point. In this example, the type MyType is used as a template argument and was defined in the mytype.h header file (MyType is needed for the instantiation of template function foo).
Line 4. Another include file inserted by the compiler. It is referenced in the function explicit instantiation at line 6 (ostream is needed for the instantiation of the operator <<).
Line 5. Code for the member function foo of class template List is going to be generated, for the specific type MyType, by this explicit instantiation.
Line 6. The operator << function has namespace scope. It is matching a template declaration in the file list.h and has its definition in list.c. The compiler inserted this explicit instantiation to force the generation of the function code.
Line 7. count function is a function that has namespace scope. The compiler inserted this explicit instantiation to force the generation of the function code.

Template-Include File (Stack.C) for the Stack class Template Example

```
/*0709395703*/#include "/home/myapp/stack.h"
/*0000000000*/#include "/home/myapp/stack.c"
template int Stack<int,20>::operator<<(int);
template int Stack<int,20>::operator>>(int &);
```

Using #pragma Directives in Header Files

When a #pragma directive is specified in a program, the directive is in effect until it is reset or overridden. There is not a specific order the compiler includes the files it created in the tempinc directory. You must reset any #pragma directives that would have an unwanted effect on other include files. For example, if you had a header file, **header1.h**, with:

```
...
#pragma options enum=small
enum enum1 {p,q,r,s};
...
```

and another file, **header2.h**, with:

```
...
enum enum2 {a,b,c,d};
...
```

enum2 would be treated as small if **header2.h** followed **header1.h**. enum2 would be treated as int if **header2.h** preceded **header1.h** and if **header2.h** had no

#pragma options enum=small directive. In this example, you should specify `#pragma options enum=reset` at the end of **header1.h** to avoid any carry over to another file.

Structuring Your Program without Automatic Template Generation

You can structure your program to define the template functions directly in your compilation units. If you know what instances of a particular template function will be needed, you can define both the template functions and the necessary declarations in one compilation unit. If you use this method, you do not have to reference any compiler-generated files. However, if you change the body of the function template, you may have to recompile many of the files. Compile and link time may be longer, and the object file produced may become quite large. To structure your program without using automatic template generation:

1. Specify the -qnotempinc option so that the compiler does not generate template-include files.
2. Place the template function definitions into one or more of your compilation units.
3. Place a reference for each template function to be generated in a compilation unit that also contains a definition of the function. For a non-member function, reference the function by including its declaration. For a member of a template class, reference the function by by explicit instantiation/specialization. Explicit instantiation forces the code generation of a template function without actually calling that function somewhere in the program. You can explicitly instantiate in namespace scope only.

In the `Stack` class template example above, the compiler generates the necessary function code if you include both **stack.h** and **stack.c** in all compilation units using instances of the `Stack` class. Code is generated for all the necessary functions. Code may be generated multiple times resulting in a very large object file.

If the tempinc feature is on, the macro `__TEMPINC__` is defined in all compilation units in which automatic template generation is on.

RELATED CONCEPTS

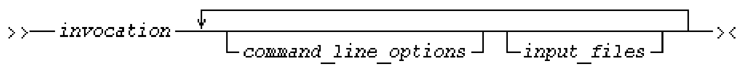
Template Implementation Model

RELATED REFERENCES

-qtempinc Batch Compiler Option

Invoke the Batch Compiler

IBM VisualAge C++ batch compilers are invoked using the following syntax, where *invocation* can be replaced with any valid VisualAge C++ invocation command:



The parameters of the compiler invocation command can be the names of input files, compiler options, and linkage-editor options. Compiler options perform a wide variety of functions, such as setting compiler characteristics, describing the object code and compiler output to be produced, and performing some preprocessor functions.

To compile without link-editing, use the “c” on page 103 compiler option. The **-c** option stops the compiler after compilation is completed and produces as output, an object file *file_name*.**o** for each *file_name*.**c** input source file, unless the -o option was used to specify a different object filename. The linkage editor is not invoked. You can link-edit the object files later using the invocation command, specifying the object files without the **-c** option.

Notes

1. Any object files produced from an earlier compilation are deleted as part of the compilation process, even if new object files are not produced.
2. By default, the invocation command calls *both* the compiler and the linkage editor. It passes linkage editor options to the linkage editor. Consequently, the invocation commands also accept all linkage editor options.

RELATED CONCEPTS

“Compiler Modes” on page 1

RELATED TASKS

“Specify Batch Compiler Options on the Command Line”

RELATED REFERENCES

“List of Batch Compiler Options and Their Defaults” on page 24
“Message Severity Levels and Compiler Response” on page 44

Specify Batch Compiler Options on the Command Line

Most options specified on the command line override both the default settings of the option and options set in the configuration file. Similarly, most options specified on the command line are in turn overridden by options set in the source file. Options that do not follow this scheme are listed in “Resolving Conflicting Compiler Options” on page 43.

There are two kinds of command-line options:

- **-q***option_keyword* (compiler-specific)
- Flag options (available to compilers on AIX systems)

-q Options

```
>>—-qoption_keyword——————————————————————————————><
                   └—=—┬—————:—————┐—┘
                       ↓           │
                       └—suboption—┘
```

Command-line options in the **-q***option_keyword* format are similar to on and off switches. If the option is specified more than once, the last instance is recognized by the compiler. For example, **-qsource** turns on the source option to produce a compiler listing; **-qnosource** turns off the source option, so no source listing is produced. For example:

```
xlC -qnosource MyFirstProg.c -qsource MyNewProg.c
```

would produce a source listing for both `MyNewProg.c` and `MyFirstProg.c` because the last **source** option specified (**-qsource**) takes precedence.

You can have multiple **-q***option_keyword* instances in the same command line, but they must be separated by blanks. Option keywords can appear in either uppercase or lowercase, but you must specify the **-q** in lowercase. You can specify any **-q***option_keyword* before or after the file name. For example:

```
xlC -qLIST -qnomaf file.c
xlC file.c -qxref -qsource
```

Some options have suboptions. You specify these with an equal sign following the **-q***option*. If the option permits more than one suboption, a colon (**:**) must separate each suboption from the next. For example:

```
xlC -qflag=w:e -qattr=full file.c
```

compiles the C source file file.c using the option **-qflag** to specify the severity level of messages to be reported, the suboptions `w` (warning) for the minimum level of severity to be reported on the listing, and `e` (error) for the minimum level of severity to be reported on the terminal. The option **-qattr** with suboption *full* will produce an attribute listing of all identifiers in the program.

Flag Options
The compilers available on AIX systems use a number of common conventional flag options. VisualAge C++ supports these flags. Lowercase flags are different from their corresponding uppercase flags. For example, **-c** and **-C** are two different compiler options: **-c** specifies that the compiler should only preprocess and compile and not invoke the linkage editor, while **-C** can be used with **-P** or **-E** to specify that user comments should be preserved.

IBM VisualAge C++ also supports flags directed to other AIX programming tools and utilities (for example, the AIX **ld** command). The compiler passes on those flags directed to **ld** at link-edit time.

Some flag options have arguments that form part of the flag. For example:

```
xlC stem.c -F/home/tools/test3/new.cfg:myc -qproclocal=sort:count
```

where new.cfg is a custom configuration file.

You can specify flags that do not take arguments in one string. For example:

```
xlC -Ocv file.c
```

has the same effect as:

```
xlC -O -c -v file.c
```

and compiles the C source file file.c with optimization (**-O**) and reports on compiler progress (**-v**), but does not invoke the linkage editor (**-c**).

A flag option that takes arguments can be specified as part of a single string, but you can only use one flag that takes arguments, and it must be the last option specified. For example, you can use the **-o** flag (to specify a name for the executable file) together with other flags, only if the **-o** option and its argument are specified last. For example:

```
xlC -Ovotest test.c
```

has the same effect as:

```
xlC -O -v -otest test.c
```

Most flag options are a single letter, but some are two letters. Note that **-pg**(extended profiling) is not the same as **-p -g** (profiling, **-p**, and generating debug information, **-g**). Take care not to specify two or more options in a single string if there is another option that uses that letter combination.

RELATED CONCEPTS

"Batch Compiler Options" on page 3

RELATED TASKS

"Invoke the Batch Compiler" on page 15
"Specify Batch Compiler Options in Your Program Source Files"
"Specify Batch Compiler Options in a Configuration File" on page 19

RELATED REFERENCES

"Resolving Conflicting Compiler Options" on page 43
"List of Batch Compiler Options and Their Defaults" on page 24

Specify Batch Compiler Options in Your Program Source Files

To specify compiler options in your program source files, use the preprocessor directive:

#pragma options *compiler_options*

If you specify more than one compiler option, separate the options using a blank space. For example:

```
#pragma options langlvl=ansi halt=s spill=1024 source
```

Most **#pragma** options directives must come before any statements in your source program; only comments, blank lines, and other **#pragma** specifications can precede them. For example, the first few lines of your program can be a comment followed by the **#pragma** options directive:

```
/* The following is an example of a #pragma options directive: */
#pragma options langlvl=ansi halt=s spill=1024 source
/* The rest of the source follows ... */
```

Options specified before any code in your source program apply to your entire program source code. You can use other **#pragma** directives throughout your program to turn an option on for a selected block of source code. For example, you can request that parts of your source code be included in your compiler listing:

```
#pragma options source
/*  Source code between the source and nosource #pragma
    options is included in the compiler listing              */
#pragma options nosource
```

Options specified in program source files override all other option settings.

These **#pragma** directives are listed in the detailed descriptions of the options to which they apply. For complete details on the various **#pragma** preprocessor directives, see the List of Pragma Preprocessor Directives.

RELATED CONCEPTS

"Batch Compiler Options" on page 3

RELATED TASKS

"Invoke the Batch Compiler" on page 15
"Specify Batch Compiler Options on the Command Line" on page 16
"Specify Batch Compiler Options in a Configuration File"

RELATED REFERENCES

"Resolving Conflicting Compiler Options" on page 43
"List of Batch Compiler Options and Their Defaults" on page 24

Specify Batch Compiler Options in a Configuration File

The default batch configuration file, /etc/vac.cfg, specifies information that the compiler uses when you invoke it. This file defines values used by the batch compiler to compile C or C++ programs. You can make entries to this file to support specific batch compilation requirements or to support other C or C++ compilation environments.

Most options specified in the batch configuration file override the default settings of the option. Similarly, most options specified in the batch configuration file are in turn overridden by options set in the source file and on the command line. Options that do not follow this scheme are listed in "Resolving Conflicting Compiler Options" on page 43.

Tailor a Configuration File
The default configuration file is /etc/vac.cfg.

You can copy this file and make changes to the copy to support specific compilation requirements or to support other C or C++ compilation environments. To specify a batch configuration file other than the default, you use the "F" on page 124 option.

For example, to make **-qnoro** the default for the **xlC** compiler invocation command, add **-qnoro** to the **xlC** stanza in your copied version of the batch configuration file.

You can link the compiler invocation command to several different names. The name you specify when you invoke the compiler determines which stanza of the batch configuration file the compiler uses. You can add other stanzas to your copy of the configuration file to customize your own compilation environment. You can use the "F" on page 124 option with the compiler invocation command to make links to select additional stanzas or to specify a stanza or another configuration file. For example:

```
xlC myfile.c -Fmyconfig:SPECIAL
```

would compile myfile.c using the SPECIAL stanza in a myconfig.cfg configuration file that you had created.

Configuration File Attributes

A stanza in the batch configuration file can contain the following attributes:

as	Path name to be used for the assembler. The default is **/bin/as**.
asopt	List of options for the assembler and not for the compiler. These override all normal processing by the compiler and are directed to the assembler specified in the **as** stanza. The string is formatted for the AIX **getopt()** subroutine as a concatenation of flag letters, with a letter followed by a colon (**:**) if the corresponding flag takes a parameter.
cppcode	Path name to be used for the code generation phase of the compiler. The default is **/usr/vacpp/exe/xlCcode**.
ccomp	C Front end. The default is **/usr/vacpp/exe/xlcentry**.
codeopt	List of options for the code-generation phase of the compiler.
comp	C++ Front end. The default is **/usr/vacpp/exe/xlCentry**.
cppopt	List of options for the lexical analysis phase of the compiler.
crt	Path name of the object file passed as the first parameter to the linkage editor. If you do not specify either the "p" on page 196 or the "pg" on page 201 option, the **crt** value is used. The default is **/lib/crt0.o**.
csuffix	Suffix for source programs. The default is **c** (lowercase c).
dis	Path name of the disassembler. The default is **/usr/vacpp/exe/dis**.
gcrt	Path name of the object file passed as the first parameter to the linkage editor. If you specify the "pg" on page 201 option, the gcrt value is used. The default is **/lib/grt0.o**.
ld	Path name to be used to link C or C++ programs. The default is **/bin/ld**.
ldopt	List of options that are directed to the linkage editor part of the compiler. These override all normal processing by the compiler and are directed to the linkage editor. If the corresponding flag takes a parameter, the string is formatted for the Aix **getopt()** subroutine as a concatenation of flag letters, with a letter followed by a colon (**:**).
libraries2	Library options, separated by commas, that the compiler passes as the last parameters to the linkage editor. **libraries2** specifies the libraries that the linkage editor is to use at link-edit time for both profiling and nonprofiling. The default is empty.
mcrt	Path name of the object file passed as the first parameter to the linkage editor if you have specified the "p" on page 196 option. The default is **/lib/mcrt0.o**.

options	A string of option flags, separated by commas, to be processed by the compiler as if they had been entered on the command line.
osuffix	The suffix for object files. The default is **.o**.
proflibs	Library options, separated by commas, that the compiler passes to the linkage editor when profiling options are specified. **proflibs** specifies the profiling libraries used by the linkage editor at link-edit time. The default is **"L" on page 159/lib/profiled** and **"L" on page 159/usr/lib/profiled**.
ssuffix	The suffix for assembler files. The default is **.s**.
use	Values for attributes are taken from the named stanza and from the local stanza. For single-valued attributes, values in the **use** stanza apply if no value is provided in the local, or default, stanza. For comma-separated lists, the values from the use stanza are added to the values from the local stanza.
xlC	The path name of the **xlC** compiler component. The default is **/usr/vacpp/bin/xlC**.

RELATED CONCEPTS

"Batch Compiler Options" on page 3

RELATED TASKS

"Invoke the Batch Compiler" on page 15
"Specify Batch Compiler Options on the Command Line" on page 16
"Specify Batch Compiler Options in Your Program Source Files" on page 18

RELATED REFERENCES

"Resolving Conflicting Compiler Options" on page 43
"List of Batch Compiler Options and Their Defaults" on page 24

Specify Batch Compiler Options for Architecture-Specific, 32- or 64-bit Compilation

You can use IBM VisualAge C++ compiler options to optimize compiler output for use on specific processor architectures. You can also instruct the compiler to compile in either 32- or 64-bit mode.

The compiler evaluates compiler options in the following order, with the last allowable one found determining the compiler mode:

1. Internal default (32-bit mode)
2. OBJECT_MODE environment variable setting, as follows:

OBJECT_MODE Setting	**User-selected compilation-mode behavior, unless overridden by configuration file or command-line options**

OBJECT_MODE Setting	User-selected compilation-mode behavior, unless overridden by configuration file or command-line options
not set	32-bit compiler mode.
32	32-bit compiler mode.
64	64-bit compiler mode.
32_64	Fatal error and stop with following message,`1501-054 OBJECT_MODE=32_64 is not a valid setting for the compiler` unless an explicit configuration file or command-line compiler-mode setting exists.
any other	Fatal error and stop with following message, `1501-055 OBJECT_MODE setting is not recognized and is not a valid setting for the compiler` unless an explicit configuration file or command-line compiler-mode setting exists.

3. Configuration file settings
4. Command line compiler options ("32, 64" on page 90, "32, 64" on page 90, "arch" on page 96, "tune" on page 230)
5. Source file statements (**#pragma options tune=***suboption*)

The compilation mode actually used by the compiler depends on a combination of the settings of the "32, 64" on page 90, "32, 64" on page 90, "arch" on page 96, and "tune" on page 230 compiler options, subject to the following conditions:

- *Compiler mode* is set acording to the last-found instance of the **-q32** or **-q64** compiler options. If neither of these compiler options is chosen, the compiler mode is set by the value of the OBJECT_MODE environment variable.
- *Architecture target* is set according to the last-found instance of the **-qarch** compiler option, provided that the specified **-qarch** setting is compatible with the *compiler mode* setting. If the **-qarch** option is not set, the compiler assumes a **-qarch** setting of **com**.
- Tuning of the architecture target is set according to the last-found instance of the **-qtune** compiler option, provided that the **-qtune** setting is compatible with the *architecture target* and *compiler mode* settings. If the **-qtune** option is not set, the compiler assumes a default **-qtune** setting according to the **-qarch** setting in use.

Allowable combinations of these options are found in the **Acceptable Compiler Mode and Processor Architecture Combinations** table.

Possible option conflicts and compiler resolution of these conflicts are described below:

- "32, 64" on page 90 or "32, 64" on page 90 setting is incompatible with user-selected "arch" on page 96 option.

  Resolution: **-q32** or **-q64** setting overrides **-qarch** option; compiler issues a warning message, sets **-qarch** option to **com**, and sets "tune" on page 230 option to the **-qarch** setting's default **-qtune** value.
- "32, 64" on page 90 or "32, 64" on page 90 setting is incompatible with user-selected "tune" on page 230 option.

  Resolution: **-q32** or **-q64** setting overrides **-qtune** option; compiler issues a warning message, and sets "tune" on page 230 option to the **-qarch** setting's default **-qtune** value.
- "arch" on page 96 option is incompatible with user-selected "tune" on page 230 option.

Resolution: Compiler issues a warning message, and sets **-qtune** tothe **-qarch** setting's default **-qtune** value.

- Selected "arch" on page 96 or "tune" on page 230 options are not known to the compiler.

  Resolution: Compiler issues a warning message, sets **-qarch** to **com**, and sets **-qtune** to the **-qarch** setting's default **-qtune** setting. The compiler mode (32- or 64-bit) is determined by the OBJECT_MODE environment variable or "32, 64" on page 90/"32, 64" on page 90 compiler settings.

RELATED CONCEPTS

"Batch Compiler Options" on page 3

RELATED TASKS

"Invoke the Batch Compiler" on page 15
"Set Up the Batch Compilation Environment" on page 7
"Specify Batch Compiler Options in Your Program Source Files" on page 18
"Specify Batch Compiler Options in a Configuration File" on page 19

RELATED REFERENCES

Acceptable Compiler Mode and Processor Architecture Combinations
"Resolving Conflicting Compiler Options" on page 43
"List of Batch Compiler Options and Their Defaults" on page 24

Invoke the Linkage Editor

The linkage editor link-edits all of the specified object files to create one executable file. Invoking the compiler with one of the invocation commands automatically calls the linkage editor unless you specify one of the following compiler options: "E" on page 116, "P" on page 195, "c" on page 103, "S" on page 213, "syntaxonly (C Only)" on page 225 or "#" on page 89.

Input Files
Object files, library files, and unstripped executable files serve as input to the linkage editor.

Object Files
Object files must have a **.o** suffix, for example, **year.o**.

Library Files
Static library file names have a **.a** suffix, for example, **libold.a**. Dynamic library file names have a **.so** suffix, for example, **libold.so**. Library files are created by combining one or more files into a single archive file with the AIX **ar** command. For a description of the **ar** command, refer to the *AIX Version 4 Commands Reference*.

Output Files
The linkage editor generates an *executable file* and places it in your current directory. The default name for an executable file is **a.out**. To name the executable file explicitly, use the "o" on page 190*file_name* option with the **xlC** command, where *file_name* is the name you want to give to the executable file. If you use the "mkshrobj" on page 184 option to create a shared library, the default name of the shared object created is shr.o.

Using the ld Command
You can invoke the linkage editor explicitly with the **ld** command. However, the compiler invocation commands set several linkage-editor options, and link some standard files into the executable output by default. In most cases, it is better to use one of the compiler invocation commands to link-edit your **.o** files.

Note: When link-editing **.o** files, *do not* use the **-e** option of the **ld** command. The default entry point of the executable output is `__start`. Changing this label with the **-e** flag can cause erratic results.

RELATED TASKS

"Invoke the Batch Compiler" on page 15

RELATED REFERENCES

"Options that Specify Linkage Options" on page 42

List of Batch Compiler Options and Their Defaults

This page lists all VisualAge C++ batch compiler options, specifying each compiler option's type and if it exists, default value. Where a * appears beside the default value for a compiler option, see the description for that option for special notes regarding the default value.

To get detailed information on any option listed, click on the that option's name in the table.

The compiler options pages describe each of the compiler options, including:
- The command-line syntax of the compiler option. The first line under the **Syntax** heading specifies the command-line or configuration-file method of specification. The second line, if one appears, is the **#pragma options** keyword for use in your source file.
- The default setting of the option if you do not specify the option on the command line, in the configuration file, or in a **#pragma** directive within your program.
- The purpose of the option and additional information about its behavior.

Uppercase letters in the option, suboption, or **#pragma options** keyword syntax represent its valid abbreviation. For example, both of the following are acceptable specifications of the **LANGlvl** option in a source file:

```
#pragma options lang=ansi
#pragma options langlvl=ansi
```

Options that appear entirely in lowercase must be entered in full.

Option Name	Type	Default	Description
"+" on page 89	*-flag*	-	Compiles any file, *filename.nnn*, as a C++ language file, where *nnn* is any suffix other than **.o**, **.a**, or **.s**.
"#" on page 89	*-flag*	-	Traces the compilation without doing anything.

“32, 64” on page 90	**-q***opt*	32	Selects 32- or 64-bit compiler mode.
Option Name	**Type**	**Default**	**Description**
“aggrcopy” on page 91	*-opt*	See “aggrcopy” on page 91.	Enables destructive copy operations for structures and unions.
“alias” on page 91	**-q***opt*	See “alias” on page 91.	Specifies which type-based aliasing is to be used during optimization.
“align” on page 93	**-q***opt*	align=full	Specifies what aggregate alignment rules the compiler uses for file compilation.
“alloc” on page 95	**-q***opt*	-	Substitutes inline code for calls to function **alloca** as if **#pragma alloca** directives are in the source code.
“ansialias” on page 95	**-q***opt*	ansialias*	Specifies whether type-based aliasing is to be used during optimization.
“arch” on page 96	**-q***opt*	arch=com	Specifies the architecture on which the executable program will be run.
“assert” on page 98	**-q***opt*	noassert	Requests the compiler to apply aliasing assertions to your compilation unit.
“attr” on page 99	**-q***opt*	noattr	Produces a compiler listing that includes an attribute listing for all identifiers.
Option Name	**Type**	**Default**	**Description**
“B” on page 99	*-flag*	-	Determines substitute path names for the compiler, assembler, linkage editor, and preprocessor.
“bitfields” on page 100	*-flag*	unsigned	Specifies if bitfields are signed.
“bmaxdata” on page 101	*-flag*	0	Sets the size of the heap in bytes.

“brtl” on page 101	*-flag*	-	Enables runtime linking.
“bstatic, bdynamic, bshared” on page 102	*-flag*	bdynamic	Instructs the linker to process subsequent shared objects as either dynamic (shared) or static.
Option Name	**Type**	**Default**	**Description**
“C” on page 103	*-flag*	-	Preserves comments in preprocessed output.
“c” on page 103	*-flag*	-	Instructs the compiler to pass source files to the compiler only.
“cache” on page 104	**-q***opt*	-	Specify a cache configuration for a specific execution machine.
“chars” on page 107	**-q***opt*	chars=unsigned	Instructs the compiler to treat all variables of type **char** as either **signed** or **unsigned**.
“check” on page 106	**-q***opt*	nocheck	Generates code which performs certain types of run-time checking.
▶ C++ -qcincl=path	**-q***opt*	nocincl	Includes files from /user/include by inserting `extern "C"` { before each <prefix> and inserting } after it.
“compact” on page 108	**-q***opt*	nocompact	When used with optimization, reduces code size where possible, at the expense of execution speed.
▶ C “cpluscmt” on page 108	**-q***opt*	nocpluscmt	Use this option if you want C++ comments to be recognized in C source files.
Option Name	**Type**	**Default**	**Description**
“D” on page 112	*-flag*	-	Defines the identifier *name* as in a **#define** preprocessor directive.
“datalocal, dataimported” on page 113	**-q***opt*	dataimported	Mark data as local or imported.

"dbxextra" on page 114	**-q***opt*	nodbxextra	Specifies that all **typedef** declarations, **struct**, **union**, and **enum** type definitions are included for **xldb** processing.
"digraph" on page 114	**-q***opt*	see "digraph" on page 114	Allows use of digraph character sequences in your program.
"dollar" on page 115	**-q***opt*	nodollar	Allows the $ symbol to be used in the names of identifiers.
dpcl	**-q***opt*	nodpcl	Generates block scopes to support the IBM Dynamic Probe Class Library.
Option Name	**Type**	**Default**	**Description**
"E" on page 116	*-flag*	-	Runs the source files named in the compiler invocation through the preprocessor.
e <*name*>	*-flag*	-	Specifies the entry name for the shared object. Equivalent to using **ld** -e *name*. See your system documentation for additional information about **ld** options.
"eh (C++ Only)" on page 117	**-q***opt*	eh	Controls exception handling.
"enum" on page 118	**-q***opt*	enum=int	Specifies the amount of storage occupied by the enumerations.
expfile	**-q***opt*	-	Saves all exported symbols in a file.
"extchk" on page 124	**-q***opt*	noextchk	Generates bind-time type checking information and checks for compile-time consistency.
Option Name	**Type**	**Default**	**Description**
"F" on page 124	*-flag*	-	Names an alternative configuration file for **xlC**.

“f” on page 125	-*flag*	-	Names a file to store a list of object files.
“fdpr” on page 126	**-q***opt*	nofdpr	Collect program information for use with the AIX **fdpr** performance-tuning utility.
“flag” on page 126	**-q***opt*	flag=i:i	Specifies the minimum severity level of diagnostic messages to be reported.
“float” on page 127	**-q***opt*	See “float” on page 127.	Specifies various floating point options to speed up or improve the accuracy of floating point operations.
“flttrap” on page 131	**-q***opt*	noflttrap	Generates extra instructions to detect and trap floating point exceptions.
“fold” on page 133	**-q***opt*	fold	Specifies that constant floating point expressions are to be evaluated at compile time.
“fullpath” on page 133	**-q***opt*	nofullpath	Specifies what path information is stored for files when you use **-g** and the distributed graphical debugger.
“funcsect” on page 134	**-q***opt*	nofuncsect	.
Option Name	**Type**	**Default**	**Description**
“G” on page 134	-*flag*	-	Linkage editor (**ld** command) option only. Used to generate a dynamic libary file.
“g” on page 135	-*flag*	-	Generates debugging information used by a debugger such as the Distributed Debugger.
“genpcomp” on page 136	**-q***opt*	nogenpcomp	Generates a precompiled version of any header file for which the original source is used.

“genproto” on page 137	**-q***opt*	nogenproto	Produces ANSI prototypes from K&R function definitions.
Option Name	**Type**	**Default**	**Description**
“halt” on page 138	**-q***opt*	halt=s	Instructs the compiler to stop after the compilation phase when it encounters errors of specified *severity* or greater.
▶ C++ “haltonmsg” on page 138	**-q***opt*	-	Instructs the compiler to stop after the compilation phase when it encounters a specific error message.
“heapdebug” on page 139	**-q***opt*	noheapdebug	Enables debug versions of memory management functions.
“hsflt” on page 140	**-q***opt*	nohsflt	Speeds up calculations by removing range checking on single-precision **float** results and on conversions from floating point to integer.
“hssngl” on page 141	**-q***opt*	nohssngl	Specifies that single-precision expressions are rounded only when the results are stored into **float** memory locations.
Option Name	**Type**	**Default**	**Description**
“I” on page 141	*-flag*	-	Specifies an additional search path if the file name in the **#include** directive is not specified using its absolute path name.
“idirfirst” on page 142	**-q***opt*	noidirfirst	Specifies the search order for files included with the **#include “***file_name***”** directive.
“ignerrno” on page 143	**-q***opt*	noignerrno	Allows the compiler to perform optimizations that assume **errno** is not modified by system calls.

Option Name	Type	Default	Description
“ignprag” on page 143	**-q***opt*	-	This option is useful for detecting aliasing pragma errors.
“info” on page 144	**-q***opt*	noinfo	Produces informational messages.
“initauto” on page 147	**-q***opt*	noinitauto	Initializes automatic storage to the two-digit hexadecimal byte value *hex_value*.
“inlglue” on page 147	**-q***opt*	noinlglue	Generates fast external linkage by inlining the pointer glue code necessary to make a call to an external function or a call through a function pointer.
“inline” on page 148	**-q***opt*	See “inline” on page 148.	Attempts to inline functions instead of generating calls to a function.
▶ C “ipa (C Only)” on page 151	**-q***opt*	object (*compile-time*), noipa (*link-time*)	Turns on or customizes a class of optimizations known as interprocedural analysis (IPA).
“isolated_call” on page 158	**-q***opt*	-	Specifies functions in the source file that have no side effects.
▶ C++ “keyword” on page 159	-q*opt*	See “keyword” on page 159	Controls whether a specified string is treated as a keyword or an identifier.
“L” on page 159	*-flag*	See “L” on page 159.	Searches the specified directory for library files specified by the “l” on page 160 option.
“l” on page 160	*-flag*	See “l” on page 160.	Searches a specified library for linking.
“langlvl” on page 160	**-q***opt*	langlvl=ansi*	Selects the C or C++ language level for compilation.
“ldbl128, longdouble” on page 173	**-q***opt*	noldbl128	Increases the size of **long double** type from 64 bits to 128 bits.

“libansi” on page 174	**-q***opt*	nolibansi	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
“linedebug” on page 174	**-q***opt*	nolinedebug	Generates abbreviated line number and source file name information for the debugger.
“list” on page 175	**-q***opt*	nolist	Produces a compiler listing that includes an object listing.
listopt	**-q***opt*	nolistopt	Produces a compiler listing that displays all options in effect.
“longlit” on page 176	**-q***opt*	nolonglit	Makes unsuffixed literals the long type for 64-bit mode.
“longlong” on page 176	**-q***opt*	longlong*	Allows **long long** types in your program.
Option Name	**Type**	**Default**	**Description**
“M” on page 177	*-flag*	-	Creates an output file that contains targets suitable for inclusion in a description file for the AIX **make** command.
“ma” on page 178	*-flag*	-	Substitutes inline code for calls to function **alloca** as if **#pragma alloca** directives are in the source code.
“macpstr” on page 178	**-q***opt*	nomacpstr	Converts Pascal string literals into null-terminated strings where the first byte contains the length of the string.
“maf” on page 181	**-q***opt*	maf	Specifies whether the floating-point multiply-add instructions are to be generated.
“makedep” on page 182	**-q***opt*	-	Creates an output file that contains targets suitable for inclusion in a description file for the AIX **make** command.

"maxerr" on page 123	**-q***opt*	nomaxerr	Instructs the compiler to halt compilation when a specified number of errors of specified or greater severity is reached.
"maxmem" on page 183	**-q***opt*	▶ C maxmem=2048 ▶ C++ maxmem=8196	Limits the amount of memory used for local tables of specific, memory-intensive optimizations.
"mbcs, dbcs" on page 184	**-q***opt*	nombcs	Use the **-qmbcs** option if your program contains multibyte characters.
"mkshrobj" on page 184	**-q***opt*	-	Creates a shared object from generated object files.
Option Name	**Type**	**Default**	**Description**
▶ C++ "namemangling (C++)" on page 189	**-q***opt*	namemangling=ansi	Selects the name mangling scheme for external symbol names generated from C++ source code.
nans	**-q***opt*	-	A floating point option.
"noprint" on page 189	**-q***opt*	-	Suppresses listings.
Option Name	**Type**	**Default**	**Description**
"O, optimize" on page 192	**-q***opt*, *-flag*	nooptimize	Optimizes code at a choice of levels during compilation.
"o" on page 190	*-flag*	-	Specifies a name or directory for the output executable file(s) created either by the compiler or the linkage editor.
"objmodel" on page 191	**-q***opt*	compat	Sets the type of object model.
"once (C Only)" on page 191	**-q***opt*	noonce	Avoids including a header file more than once even if it is specified in several of the files you are compiling.

Option Name	Type	Default	Description
"P" on page 195	*-flag*	-	Preprocesses the C or C++ source files named in the compiler invocation and creates an output preprocessed source file for each input source file.
"p" on page 196	*-flag*	-	Sets up the object files produced by the compiler for profiling.
"pascal" on page 197	**-q***opt*	nopascal	Ignores the word **pascal** in type specifiers and function declarations.
"path" on page 197	**-q***opt*	-	.
"pdf1, pdf2" on page 198	**-q***opt*	nopdf1 nopdf2	Tunes optimizations through Profile-Directed Feedback.
"pg" on page 201	*-flag*	-	Sets up the object files for profiling, but provides more information than is provided by the "p" on page 196 option.
"phsinfo" on page 201	**-q***opt*	nophsinfo	Reports the time taken in each compilation phase.
"priority" on page 202	**-q***opt*	-	Specifies the priority level for the initialization of static constructors
"proclocal, procimported, procunknown" on page 202	**-q***opt*	See "proclocal, procimported, procunknown" on page 202	Mark functions as local, imported, or unknown.
"profile" on page 204	**-q***opt*	noprofile	Sets up the object files for profiling. Suboptions indicate the profiling tool.
"proto" on page 205	**-q***opt*	noproto	Assumes all functions are prototyped.
Option Name	**Type**	**Default**	**Description**

“Q” on page 206	*-flag*	See “Q” on page 206.	Attempts to inline functions instead of generating calls to a function.

Option Name	Type	Default	Description
“r” on page 209	*-flag*	-	Produces a relocatable object.
“rndsngl” on page 209	**-q***opt*	norndsngl	Specifies that the result of each single-precision (**float**) operation is to be rounded to single precision.
“ro” on page 210	**-q***opt*	ro*	Specifies the storage type for string literals.
“roconst” on page 210	**-q***opt*	roconst*	Specifies the storage location for constant values.
“rrm” on page 211	**-q***opt*	norrm	Prevents floating-point optimizations that are incompatible with run-time rounding to plus and minus infinity modes.
“rtti” on page 212	**-q***opt*	nortti	Generates run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.

Option Name	Type	Default	Description
“S” on page 213	*-flag*	-	Generates an assembly language file (**.s**) for each source file.
“s” on page 214	*-flag*	-	Strips symbol table.
“showinc” on page 214	**-q***opt*	noshowinc	If used with “source” on page 217, all the include files are included in the source listing.
▶ C “smp (C Only)” on page 215	**-q***opt*	nosmp	Specifies if and how parallelized object code is generated.
“source” on page 217	**-q***opt*	nosource	Produces a compiler listing and includes source code.

“spill” on page 218	**-q***opt*	spill=512	Specifies the size of the register allocation spill area.
“spnans” on page 218	**-q***opt*	nospnans	Generates extra instructions to detect signalling NaN on conversion from single precision to double precision.
“srcmsg (C Only)” on page 219	**-q***opt*	nosrcmsg	Adds the corresponding source code lines to the diagnostic messages in the **stderr** file.
“staticinline” on page 219	**-q***opt*	nostaticinline	Controls whether inline functions are treated as static or extern.
“statsym” on page 220	**-q***opt*	nostatsym	Adds user-defined, non-external names that have a persistent storage class to the name list.
“stdinc” on page 220	**-q***opt*	stdinc	Specifies which files are included with **#include** <*file_name*> and **#include** “*file_name*” directives.
“strict” on page 221	**-q***opt*	See “strict” on page 221	Turns off aggressive optimizations of the “O, optimize” on page 192 option that have the potential to alter the semantics of your program.
“strict_induction” on page 222	**-q***opt*	See “strict_induction” on page 222	Disables loop induction variable optimizations that have the potential to alter the semantics of your program.
“suppress” on page 223	**-q***opt*	nosuppress	Specifies compiler message numbers to be suppressed.
“symtab” on page 223	**-q***opt*	-	Set symbol tables for unreferenced variables or xcoff objects.
“syntaxonly (C Only)” on page 225	**-q***opt*	-	Causes the compiler to perform syntax checking without generating an object file.
Option Name	**Type**	**Default**	**Description**

"t" on page 227	*-flag*	See "t" on page 227.	Adds the prefix specified by the "B" on page 99 option to designated programs. -tE replaces the CreateExportList script.
"tabsize" on page 225	**-q***opt*	tabsize=8	Changes the length of tabs as perceived by the compiler.
"tbtable" on page 226	**-q***opt*	tbtable=full*	Sets traceback table characteristics.
"tempinc" on page 228	**-q***opt*	See "tempinc" on page 228.	Generates separate include files for template functions and class declarations, and places these files in a directory which can be optionally specified.
"tempmax" on page 228	**-q***opt*	tempmax=1	Specifies the maximum number of template include files to be generated by the "tempinc" on page 228 option for each header file.
"threaded" on page 229	**-q***opt*	See "threaded" on page 229.	Indicates that the program will run in a multi-threaded environment.
tmplparse	**-q***opt*	tmplparse=no	Controls whether parsing and semantic checking are applied to template definition implementations.
"tune" on page 230	**-q***opt*	See "tune" on page 230.	Specifies the architecture for which the executable program is optimized.
"twolink" on page 231	**-q***opt*	notwolink*	Minimizes the number of static constructors included from libraries.
Option Name	**Type**	**Default**	**Description**
"U" on page 232	*-flag*	-	Undefines a specified identifier defined by the compiler or by the "D" on page 112 option.

“unique” on page 233	**-q***opt*	nounique	Generates unique names for static constructor/deconstructor file compilation units.
“unroll” on page 234	**-q***opt*	unroll=4*	Unrolls inner loops in the program by a specified factor.
“upconv” on page 235	**-q***opt*	noupconv*	Preserves the **unsigned** specification when performing integral promotions.
“usepcomp” on page 236	**-q***opt*	nousepcomp	Use precompiled header files for any files that have not changed since the precompiled header was created.
Option Name	**Type**	**Default**	**Description**
“V” on page 237	*-flag*	-	Instructs the compiler to report information on the progress of the compilation in a command-like format.
“v” on page 238	*-flag*	-	Instructs the compiler to report information on the progress of the compilation.
“vftable” on page 238	*-flag*	See vftable.	Controls the generation of virtual function tables.
Option Name	**Type**	**Default**	**Description**
“W” on page 239	*-flag*	-	Passes the listed words to a designated compiler program.
“w” on page 240	*-flag*	-	Requests that warning messages be suppressed.
“warn64” on page 240	**-q***opt*	nowarn64	Enables warning of possible long to integer data truncations.
Option Name	**Type**	**Default**	**Description**
“xcall” on page 241	**-q***opt*	noxcall	Generates code to static routines within a compilation unit as if they were external calls.

Option Name	Type	Default	Description
“xref” on page 241	**-q***opt*	noxref	Produces a compiler listing that includes a cross-reference listing of all identifiers.
“y” on page 242	*-flag*	-	Specifies the compile-time rounding mode of constant floating-point expressions. See also rndflt.
“Z” on page 243	*-flag*	-	Specifies a search path for library names.

* beside the default value for a compiler option indicates that you should see the description for that option for special notes regarding the default value.

RELATED REFERENCES

“Options that Specify Compiler Characteristics”
“Options that Specify Debugging Features” on page 39
“Options that Specify Preprocessor Options” on page 40
“Options that Specify Compiler Output” on page 40
“Options that Specify the Compiler Object Code Produced” on page 41
“Options that Specify Linkage Options” on page 42
“Resolving Conflicting Compiler Options” on page 43
Equivalent Batch Compile-Link and Incremental Build Options

Options that Specify Compiler Characteristics

To:	See:
Specify the language level	“langlvl” on page 160
▶ C++ Specify whether a string is treated as a keyword or an identifier whenever it appears in the C++ source.	“keyword” on page 159
Specify a different configuration file or stanza	“F” on page 124
Specify path names to other program names	“B” on page 99, “t” on page 227, “path” on page 197
Specify program options	“W” on page 239,
Specify a search path	“I” on page 141
Specify if **char** variables are treated as **signed** or **unsigned**	“chars” on page 107
Specify if bitfields are signed.	“bitfields” on page 100
Specify the use of multibyte characters	“mbcs, dbcs” on page 184

Change the length of tabs in your source file	“tabsize” on page 225

RELATED REFERENCES

“List of Batch Compiler Options and Their Defaults” on page 24
“Options that Specify Debugging Features”
“Options that Specify Preprocessor Options” on page 40
“Options that Specify Compiler Output” on page 40
“Options that Specify the Compiler Object Code Produced” on page 41
“Options that Specify Linkage Options” on page 42
Equivalent Batch Compile-Link and Incremental Build Options

Options that Specify Debugging Features

To:	See:
Produce only line number and source file name information	“linedebug” on page 174
Produce debug information	“g” on page 135
Enable debug versions of memory management functions	“heapdebug” on page 139
Specify full path information when you use “g” on page 135	“fullpath” on page 133
Generate and set the charcateristics of the traceback table	“tbtable” on page 226
Include all typedef declarations, struct, union, and enum type definitions for xldb processing.	“dbxextra” on page 114
Produce informational messages.	“info” on page 144
Initialize automatic storage to a two-digit hexadecimal byte value.	“initauto” on page 147
Generate extra instructions to detect and trap floating point exceptions.	“flttrap” on page 131
Generate bind-time type checking information and check for compile-time consistency.	“extchk” on page 124
Specify minimum severity level of diagnostic messages to be reported.	“flag” on page 126
Set up object files for profiling	“profile” on page 204 “p” on page 196 “pg” on page 201
Trap null pointer usage, subscript exceeding array bounds, or division of an integer by zero	“check” on page 106
Ignore “isolated_call” on page 158 aliasing pragmas	“ignprag” on page 143

RELATED CONCEPTS

Debugging Memory Heaps
Memory Management Functions
Managing Memory with Multiple Memory Heaps

RELATED TASKS

Debug Programs with Heap Memory

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
"Options that Specify Preprocessor Options"
"Options that Specify Compiler Output"
"Options that Specify the Compiler Object Code Produced" on page 41
"Options that Specify Linkage Options" on page 42
Equivalent Batch Compile-Link and Incremental Build Options

Options that Specify Preprocessor Options

To:	See:
Define a name in a **#define** directive	"D" on page 112
Undefine a name as in a **#undefine** directive	"U" on page 232
Make unsuffixed literals into the long type in 64-bit mode.	"longlit" on page 176
Create an output file for use with the **make** command	"M" on page 177 "makedep" on page 182

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
"Options that Specify Debugging Features" on page 39
"Options that Specify Compiler Output"
"Options that Specify the Compiler Object Code Produced" on page 41
"Options that Specify Linkage Options" on page 42
Equivalent Batch Compile-Link and Incremental Build Options

Options that Specify Compiler Output

To:	See:
Perform syntax checking but do not generate an object file	"syntaxonly (C Only)" on page 225
Preprocess but not compile	"C" on page 103 "E" on page 116 "P" on page 195
Create assembler source but not link	"S" on page 213
Compile but not link	"c" on page 103
Create a dynamic library object file	"G" on page 134
Produce compiler listings	"noprint" on page 189 "source" on page 217 "showinc" on page 214 "srcmsg (C Only)" on page 219 "xref" on page 241 "attr" on page 99 "list" on page 175 listopt

Specify severity level of diagnostic messages	"flag" on page 126
Suppress specific messages	"suppress" on page 223
Suppress warning messages	"w" on page 240
Halt the compiler output if errors of specified *severity* or greater are encountered	"halt" on page 138
Halt the compiler output if *num* errors of specified or greater *severity* are encountered	"maxerr" on page 123
Halt the compiler output a particular message is encountered	"haltonmsg" on page 138
Produce information messages	"info" on page 144
Report the time taken for compilation	"phsinfo" on page 201
Report status information as the compilation proceeds	"v" on page 238, "V" on page 237
Preview the compilation	"#" on page 89

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
"Options that Specify Debugging Features" on page 39
"Options that Specify Preprocessor Options" on page 40
"Options that Specify the Compiler Object Code Produced"
"Options that Specify Linkage Options" on page 42
Equivalent Batch Compile-Link and Incremental Build Options

Options that Specify the Compiler Object Code Produced

To:	See:
Specify the architecture on which the executable program will be run	"32, 64" on page 90 "arch" on page 96 "tune" on page 230
Use the linkage editor to create a dynamic library file (**ld** command only)	"G" on page 134
Specify the register allocation spill area	"spill" on page 218
Specify if and how parallelized object code is produced (C Only)	"smp (C Only)" on page 215
Choose code optimization options	"O, optimize" on page 192 "pdf1, pdf2" on page 198 "ipa (C Only)" on page 151 "unroll" on page 234
Generate information used by the **fdpr** performance-tuning utility	"fdpr" on page 126
Set inlining options	"Q" on page 206 "ipa (C Only)" on page 151 "inline" on page 148
Choose alignment rules for aggregates	"align" on page 93
Choose storage type for constant values	"roconst" on page 210
Choose storage types for string literals	"ro" on page 210
Set the size of a **long double** (64 or 128 bits)	"ldbl128, longdouble" on page 173

Ignore **long long int** types	“longlong” on page 176
Set the rounding mode of floating-point expressions	“y” on page 242
Reduce code size	“compact” on page 108
Set object model	“objmodel” on page 191
Set floating point options	“float” on page 127
Set rounding of single-precision expressions	“float” on page 127 (**-qfloat=hssngl**)
Include extra instructions to detect NaN	“float” on page 127 (**-qfloat=nans**)
Remove range checking	“float” on page 127 (**-qfloat=hsflt**)
Set rounding of single-precision (**float**) operations	“float” on page 127 (**-qfloat=rndsngl**)
Detect and trap floating point exceptions	“flttrap” on page 131
Generate floating point multiply-add instructions	“float” on page 127 (**-qfloat=maf**)
Prevent incompatible optimizations	“float” on page 127 (**-qfloat=rrm**)
Evaluate floating point expressions at compile time	“float” on page 127 (**-qfloat=fold**)
Generate bind-time type checking	“extchk” on page 124
Choose type-based aliasing during optimization	“alias” on page 91 “ansialias” on page 95
Initialize automatic storage	“initauto” on page 147
Limit the amount of memory	“maxmem” on page 183
Mark data as local or imported	“datalocal, dataimported” on page 113
Mark functions as local, imported, or unknown	“proclocal, procimported, procunknown” on page 202 “ma” on page 178
Substitute inline code for calls to **alloca**	“ma” on page 178
Perform optimizations that assume **errno** is not modified by system calls	“ignerrno” on page 143

RELATED REFERENCES

“List of Batch Compiler Options and Their Defaults” on page 24
“Options that Specify Compiler Characteristics” on page 38
“Options that Specify Debugging Features” on page 39
“Options that Specify Preprocessor Options” on page 40
“Options that Specify Compiler Output” on page 40
“Options that Specify Linkage Options”
Equivalent Batch Compile-Link and Incremental Build Options

Options that Specify Linkage Options

To:	See:
Name a file containing a list of object files for linking	“f” on page 125
Name the output file or directory	“o” on page 190
Search specified libraries	“l” on page 160

Search a path for libraries	“L” on page 159
Specify a search path for library names	“L” on page 159
Produce an output file even if not all symbols are resolved	“r” on page 209
Specify which types of library file are used by the linkage editor	“brtl” on page 101 “bstatic, bdynamic, bshared” on page 102
Sets the size of the heap in bytes.	“bmaxdata” on page 101
▶ C++ Selects the name mangling scheme.	“namemangling (C++)” on page 189
Generate fast external linkage	“inlglue” on page 147

There are other -b linker options. You can find documentation about them in your AIX system documentation.

RELATED REFERENCES

“List of Batch Compiler Options and Their Defaults” on page 24
“Options that Specify Compiler Characteristics” on page 38
“Options that Specify Debugging Features” on page 39
“Options that Specify Preprocessor Options” on page 40
“Options that Specify Compiler Output” on page 40
“Options that Specify the Compiler Object Code Produced” on page 41
Equivalent Batch Compile-Link and Incremental Build Options

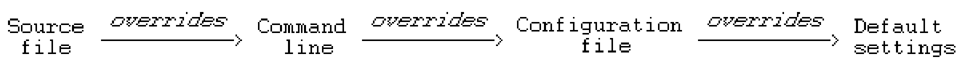
Resolving Conflicting Compiler Options

In general, if more than one variation of the same option is specified (with the exception of **xref** and **attr**), the compiler uses the setting of the last one specified. Compiler options specified on the command line must appear in the order you want the compiler to process them.

If a command-line flag is valid for more than one compiler program (for example **-B**, **-W**, or **-I** applied to the compiler, linkage editor, and assembler program names), you must specify it in **cppopt**, **codeopt**, **inlineopt**, **ldopt**, or **asopt** in the configuration file. The command-line flags must appear in the order that they are to be directed to the appropriate compiler program.

Two exceptions to the rules of conflicting options are the “I” on page 141*directory* and “L” on page 159*directory* options, which have cumulative effects when they are specified more than once.

In most cases, conflicting or incompatible options are resolved according to the precedence shown in the following figure:



Most options that do not follow this scheme are summarized in the following table.

Option	Conflicting Options	Resolution
“halt” on page 138	Severity specified	Lowest severity specified.
“noprint” on page 189	“xref” on page 241 | “attr” on page 99 | “source” on page 217 on page 217 on page 217 on page 217 | listopt | “list” on page 175	-qnoprint
“float” on page 127=rsqrt	“ignerrno” on page 143	Last option specified
“xref” on page 241	xref=FULL	xref=FULL
“attr” on page 99	attr=FULL	attr=FULL
“p” on page 196 | “pg” on page 201 | “proto” on page 205 on page 205 on page 205 on page 205	“p” on page 196 | “pg” on page 201 | “proto” on page 205 on page 205 on page 205 on page 205	Last option specified
“hsflt” on page 140	“rndsngl” on page 209 | “spnans” on page 218	hsflt
“hssngl” on page 141	“rndsngl” on page 209 | “spnans” on page 218	hssngl
“E” on page 116	“P” on page 195 | “o” on page 190 | “S” on page 213 on page 213 on page 213	-E
“P” on page 195	“c” on page 103 | “o” on page 190 | “S” on page 213 on page 213 on page 213	-P
“#” on page 89	“v” on page 238	-#
“F” on page 124	“B” on page 99 | “t” on page 227 | “W” on page 239 on page 239 on page 239 | “path” on page 197 on page 197 | configuration file settings	-B | -t | -W | -qpath
“path” on page 197	“B” on page 99 | “t” on page 227	-qpath overrides -B and -t
“S” on page 213	“c” on page 103	-S

RELATED CONCEPTS

“Specify Batch Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 21

Message Severity Levels and Compiler Response

The following table shows the compiler response associated with each level of message severity.

Letter	Severity	Compiler Response
I	Informational	Compilation continues. The message reports conditions found during compilation.

Letter	Severity	Compiler Response
W	Warning	Compilation continues. The message reports valid, but possibly unintended, conditions.
▶ C E	Error	Compilation continues and object code is generated. Error conditions exist that the compiler can correct, but the program might not run correctly.
S	Severe error	Compilation continues, but object code is not generated. Error conditions exist that the compiler cannot correct.
U	Unrecoverable error	The compiler halts. An internal compiler error has been found. This message should be reported to your IBM service representative.

RELATED CONCEPTS

"Compiler Message and Listing Information" on page 6
"Batch Compiler Message Format" on page 46

RELATED REFERENCES

"Batch Compiler Return Codes"
"halt" on page 138
"W" on page 239

Batch Compiler Return Codes

At the end of compilation, the compiler sets the return code to zero under any of the following conditions:

- No messages are issued.
- The highest severity level of all errors diagnosed is less than the setting of the "halt" on page 138 compiler option, **and** the number of errors did not reach the limit set by the "maxerr" on page 123 compiler option. The default for -qhalt is S.
- No message specifid by the "haltonmsg" on page 138 compiler option is issued.

Otherwise, the compiler sets the return code to one of the following values:

Return Code	Error Type
1	Any error with a severity level higher than the setting of the halt compiler option has been detected.
40	An option error or an unrecoverable error has been detected.
41	A configuration file error has been detected.
250	An out-of-memory error has been detected. The **xlC** command cannot allocate any more memory for its use.
251	A signal-received error has been detected. That is, an unrecoverable error or interrupt signal has occurred.
252	A file-not-found error has been detected.
253	An input/output error has been detected: files cannot be read or written to.

254	A fork error has been detected. A new process cannot be created.
255	An error has been detected while the process was running.

RELATED CONCEPTS

"Compiler Message and Listing Information" on page 6
"Batch Compiler Message Format"

RELATED REFERENCES

"Message Severity Levels and Compiler Response" on page 44

Batch Compiler Message Format

Diagnostic messages have the following format when the "srcmsg (C Only)" on page 219 option is active (which is the default):

```
"file", line line_number.column_number: 15dd-nnn (severity) text.
```

where:

file	is the name of the C or C++ source file with the error.	
line_number	is the line number of the error.	
column_number	is the column number for the error	
15	is the compiler product identifier	
cc	is a two-digit code indicating the VisualAge C++ component that issued the message. *cc* can have the following values:	
	00	- code generating or optimizing message
	01	- compiler services message.
	05	- message specific to the C compiler
	06	- message specific to the C compiler
	40	- message specific to the C++ compiler
	47	- message specific to munch utility
	86	- message specific to interprocedural analysis (IPA).
nnn	is the message number	
severity	is a letter representing the severity of the error	
text	is a message describing the error	

C

Diagnostic messages have the following format when the "srcmsg (C Only)" on page 219 option is specified:

```
    x - 15dd-nnn(severity) text.
```

where **x** is a letter referring to a finger in the finger line.

To help you find the exact point of the error in the line, when you use the "srcmsg (C Only)" on page 219 option, a finger line is produced below the source code line if the error is applicable to a specific column in the source line. For example:

```
        10 | int add(int, int)
             ....a...b....c...
    a - 1506-166 (S) Definition of function add requires parentheses.
    b - 1506-172 (S) Parameter type list for function add contains
    parameters without identifiers.
    c - 1506-172 (S) Parameter type list for function add contains
    parameters without identifiers.
```

The finger line may also be produced in the source listing if you specify the "source" on page 217 option.

RELATED CONCEPTS

"Compiler Message and Listing Information" on page 6

RELATED REFERENCES

"Batch Compiler Return Codes" on page 45
"Message Severity Levels and Compiler Response" on page 44

National Languages Support in VisualAge C++

This and related pages summarize the national language support (NLS) specific to IBM VisualAge C++.

Note: You must specify the "mbcs, dbcs" on page 184 option to use multibyte characters anywhere in your program.

Support for multibyte characters includes support for wide characters. Generally, wide characters are permitted anywhere multibyte characters are, but they are incompatible with multibyte characters in the same string because their bit patterns differ. Wherever permitted, you can mix single-byte and multibyte characters in the same string. See Restrictions for more information.

RELATED REFERENCES

"Converting Files Containing Multibyte Data to New Code Pages"
"Multibyte Character Support" on page 48

Converting Files Containing Multibyte Data to New Code Pages

If you have installed new code pages on your system, you must use the AIX **iconv** migration utility to convert files containing multibyte data to use new code pages. This command converts files containing multibyte data from the **IBM-932** code set to the **IBM-euc** code set.

The iconv command is described in the *AIX Version 4 Commands Reference*. Using the NLS code set converters with the **iconv** command is described in "Converters Overview for Programming" in the *AIX Version 4 General Programming Concepts*.

RELATED REFERENCES

"National Languages Support in VisualAge C++" on page 47
"Multibyte Character Support"

Multibyte Character Support

In the examples that follow, *multibyte_char* represents any string of one or more multibyte characters.

String Literals and Character Constants

Multibyte characters are supported in string literals and character constants. Strings containing multibyte characters are treated in essentially the same way as strings without multibyte characters. Multibyte characters can appear in several contexts:

- Preprocessor directives
- Macro definitions
- The # and ## operators
- The definition of the macro name in the "D" on page 112 compiler option

Wide-character strings can be manipulated the same way as single-byte character strings. The system provides equivalent wide-character and single-byte string functions.

The default storage type for all string literals is read-only. The "ro" on page 210 option sets the storage type of string literals to read-only, and the "ro" on page 210 option makes string literals writable.

Note: Because a character constant can store only 1 byte, avoid assigning multibyte characters to character constants. Only the last byte of a multibyte character constant is stored. Use a wide-character representation instead. Wide-character string literals and constants must be prefixed by L. For example:

```
wchar_t *a = L"wide_char_string" ;
wchar_t b = L'c'
```

Preprocessor Directives

The following preprocessor directives permit multibyte-character constants and string literals:

- **#define**
- **#pragma comment**
- **#include**

Macro Definitions

Because string literals and character constants can be part of **#define** statements, multibyte characters are also permitted in both object-like and function-like macro definitions.

Compiler Options

Multibyte characters can appear in the compiler suboptions that take file names as arguments:

- "l" on page 160 *key*
- "o" on page 190 *file_name*

- "B" on page 99 *prefix*
- "F" on page 124 *config_file:stanza*
- "I" on page 141 *directory*
- "L" on page 159 *directory*

The "D" on page 112 *name=definition* option permits multibyte characters in the definition of the macro name. In the following example, the first definition is a string literal, and the second is a character constant:

```
-DMYMACRO="kpsmultibyte_chardcs"
-DMYMACRO='multibyte_char'
```

The "mbcs, dbcs" on page 184 compiler option permits both double-byte and multibyte characters. In other respects, it is equivalent to the "mbcs, dbcs" on page 184 option, but it should be used when multibyte characters appear in the program.

The listings produced by the "list" on page 175 and "source" on page 217 options display the date and time for the appropriate international language. Multibyte characters in the file name of the C or C++ source file also appear in the name of the corresponding list file. For example, a C source file called:

```
multibyte_char.c
```

gives a list file called

```
multibyte_char.lst
```

File Names and Comments

Any file name can contain multibyte characters. The file name can be a relative or absolute path name. For example:

```
#include<multibyte_char/mydir/mysource/multibyte_char.h>
#include "multibyte_char.h"
xlC /u/myhome/c_programs/kanji_files/multibyte_char.c
-omultibyte_char
```

Multibyte characters are also permitted in comments, if you specify the "mbcs, dbcs" on page 184 compiler option.

Restrictions

- Multibyte characters are not permitted in identifiers.
- Hexadecimal values for multibyte characters must be in the range of the code page being used.
- You cannot mix wide characters and multibyte characters in macro definitions. For example, a macro expansion that concatenates a wide string and a multibyte string is not permitted.
- Assignment between wide characters and multibyte characters is not permitted.
- Concatenating wide character strings and multibyte character strings is not permitted.

RELATED REFERENCES

"National Languages Support in VisualAge C++" on page 47
"Multibyte Character Support" on page 48
"mbcs, dbcs" on page 184 Compiler Option
"mbcs, dbcs" on page 184 Compiler Option
# Operator

Built-in Functions for PowerPC Processors

PowerPC platforms support RS/6000® machine instructions not available on other platforms. If performance is critical to your application, the VisualAge C++ compilers provide a set of built-in functions that directly map to certain PowerPC instructions. By using these functions, function call return costs, parameter passing, stack adjustment and all the additional costs related with function invocations are eliminated.

Not all functions described below are supported by all RS/6000 processors. Using an unsupported function will result in an error message being displayed.

Name	Prototype	Return Value or Action Performed
__fmadds( )	float __fmadds (float, float, float);	__fmadds (*a*, *x*, *y*) = [*a* * *x* + *y*]
__fmadd( )	double __fmadd (double, double, double);	__fmadd (*a*, *x*, *y*) = [*a* * *x* + *y*]
__fmsubs( )	float __fmsubs (float, float, float);	__fmsubs (*a*, *x*, *y*) = [*a* * *x* - *y*]
__fmsub( )	double __fmsub (double, double, double);	__fmsub (*a*, *x*, *y*) = [*a* * *x* - *y*]
__fnmadds( )	float __fnmadds (float, float, float);	__fnmadds (*a*, *x*, *y*) = [- (*a* * *x* + *y*)]
__fnmadd( )	double __fnmadd (double, double, double);	__fnmadd (*a*, *x*, *y*) = [- (*a* * *x* + *y*)]
__fnmsubs( )	float __fnmsubs (float, float, float);	__fnmsubs (*a*, *x*, *y*) = [- (*a* * *x* - *y*)]
__fnmsub( )	double __fnmsub (double double, double);	__fnmsub (*a*, *x*, *y*) = [- (*a* * *x* - *y*)]
__fsqrts( )	float __fsqrts (float);	__fsqrts (*x*) = square root of *x*
__fsqrt( )	double __fsqrt (double);	__fsqrt (*x*) = square root of *x*
__frsqrte( )	double __frsqrte (double);	__frsqrte (*x*) = [(estimate of) 1.0/sqrt(*x*)]
__fres( )	float __fres (float);	__fres (*x*) = [(estimate of) 1.0/*x*]

__fsels( )	float __fsels (float, float, float);	if (*a* >= 0.0) then __fsels (*a*, *x*, *y*) = *x*;
		else __fsels (*a*, *x*, *y*) = *y*

__fsel( )	double __fsel (double, double, double);	if (*a* >= 0.0) then __fsel (*a*, *x*, *y*) = *x*;
		else __fsel (*a*, *x*, *y*) = *y*

__fabss( )	float __fabss (float);	__fabss (*x*) = |*x*|
__fabs( )	double __fabs (double);	__fabs (*x*) = |*x*|
__fnabss( )	float __fnabss (float);	__fnabss (*x*) = -|*x*|
__fnabs( )	double __fnabs (double);	__fnabs (*x*) = -|*x*|

Name	Prototype	Return Value or Action Performed
__dcbt( )	void __dcbt (void *);	Data Cache Block Touch. Loads the block of memory containing the specified address into the data cache.
__dcbz( )	void __dcbz (void *);	Data Cache Block set to Zero. Sets the specified address in the data cache to zero (0).
__trap( )	void __trap (int);	Trap if the parameter is not zero.
__trapd( )	void __trapd (longlong);	Trap if the parameter is not zero.

Other builtin functions contained in /usr/vacpp/include/builtins.h have the following prototype:

- int __assert1(int, int, int);
- void __assert2(int);
- void __bcopy(char *, char *, int);
- void __bzero(char *, int);
- unsigned int __cnttz4(unsigned int);
- unsigned int __cnttz8(unsigned long long);
- double __exp(double);
- void __iospace_eieio(void);
- void __iospace_sync(void);
- unsigned int __load4r(unsigned int *);
- unsigned short __load2r(unsigned short *);
- int __parthds(void);
- double __pow(double, double);
- double __readflm(void);
- double __setflm(double);
- double __setrnd(int);
- void __store2r(unsigned short, unsigned short *);
- void __store4r(unsigned int, unsigned int *);
- int __usrthds(void);

RELATED REFERENCES

Acceptable Compiler Mode and Processor Architecture Combinations

Chapter 2. Program Parallelization (C Only)

The compiler offers you two methods of implementing shared memory program parallelization. These are:

- Automatic and explicit parallelization of countable loops using IBM pragma directives.
- Program parallelization using pragma directives compliant to the **OpenMP Application Program Interface** specification.

All methods of program parallelization are enabled when the **-qsmp** compiler option is in effect without the **omp** suboption. You can enable strict OpenMP compliance with the **-qsmp=omp** compiler option, but doing so will disable automatic parallelization.

Parallel regions of program code are executed by multiple threads, possibly running on multiple processors. The number of threads created is determined by the run-time options and calls to library functions. Work is distributed among available threads according to the specified scheduling algorithm.

Note: The **-qsmp** option must only be used together with thread-safe compiler invocation modes.

IBM Directives

IBM directives exploit shared memory parallelism through the parallelization of *countable loops*. A loop is considered to be *countable* if it has any of the forms described in "Countable Loops (C Only)" on page 54.

The compiler can automatically locate and where possible parallelize all countable loops in your program code. In general, a countable loop is automatically parallelized only if all of the follow conditions are met:

- the order in which loop iterations start or end does not affect the results of the program.
- the loop does not contain I/O operations.
- floating point reductions inside the loop are not affected by round-off error, unless the **-qnostrict** option is in effect.
- the **-qnostrict_induction** compiler option is in effect.
- the **-qsmp** compiler option is in effect without the **opm** suboption. The compiler must be invoked using a thread-safe compiler mode.

You can also explicitly instruct the compiler to parallelize selected countable loops.

The C for AIX compiler provides pragma directives that you can use to improve on automatic parallelization performed by the compiler. Pragmas fall into two general categories.

1. The first category of pragmas lets you give the compiler information on the characteristics of a specific countable loop. The compiler uses this information to perform more efficient automatic parallelization of the loop.

2. The second category gives you explicit control over parallelization. Use these pragmas to force or suppress parallelization of a loop, apply specific parallelization algorithms to a loop, and synchronize access to shared variables using critical sections.

OpenMP Directives

OpenMP directives exploit shared memory parallelism by defining various types of *parallel regions*. Parallel regions can include both iterative and non-iterative segments of program code.

Pragmas fall into four general categories:

1. The first category of pragmas lets you define parallel regions in which work is done by threads in parallel. Most of the OpenMP directives either statically or dynamically bind to an enclosing parallel region.
2. The second category lets you define how work will be distributed or shared across the threads in a parallel region.
3. The third category lets you control synchronization among threads.
4. The fourth category lets you define the scope of data visibility across threads.

RELATED CONCEPTS

"Countable Loops (C Only)"
"Reduction Operations in Parallelized Loops (C Only)" on page 56
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Compiler Modes" on page 1

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59
"Invoke the Batch Compiler" on page 15

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"IBM Run-time Options for Parallel Processing (C Only)" on page 63
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66
"Built-in Functions Used for Parallel Processing (C Only)" on page 61
"smp (C Only)" on page 215 Compiler Option
"strict" on page 221 Compiler Option
"strict_induction" on page 222 Compiler Option
OpenMP Specification

Countable Loops (C Only)

A loop is considered to be *countable* if it has any of the forms shown below, and:

- there is no branching into or outside of the loop.
- the *incr_expr* expression is not within a critical section.

The following are examples of countable loops.

```
for ([iv]; exit_cond; incr_expr)
 statement
```

```
for ([iv]; exit_cond; [expr] {
    [declaration_list]
    [statement_list]
    incr_expr;
    [statement_list]
}
while (exit_cond) {
    [declaration_list]
    [statement_list]
    incr_expr;
    [statement_list]
}
do {
    [declaration_list]
    [statement_list]
    incr_expr;
    [statement_list]
} while (exit_cond)
```

The following definitions apply to the above examples:

	takes form:
exit_cond	*iv* <= *ub* *iv* < *ub* *iv* >= *ub* *iv* > *ub*

	takes form:

incr_expr	`++iv`
	`iv++`
	`–iv`
	`iv–`
	`iv += incr`
	`iv -= incr`
	`iv = iv + incr`
	`iv = incr + iv`
	`iv = iv - incr`

iv	Iteration variable. The iteration variable is a signed integer that has either automatic or register storage class, does not have its address taken, and is not modified anywhere in the loop except in *incr_expr*.
incr	Loop invariant signed integer expression. The value of the expression is known at run-time and is not 0. *incr* cannot reference extern or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.
ub	Loop invariant signed integer expression. *ub* cannot reference extern or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Reduction Operations in Parallelized Loops (C Only)" on page 56

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60

Reduction Operations in Parallelized Loops (C Only)

The compiler can recognize and properly handle most reduction operations in a loop during both automatic and explicit parallelization. In particular, it can handle reduction statements that have either of the following forms:

```
var = var op expr;
var assign_op expr;
```

where:

var	Is an identifier designating an automatic or register variable that does not have its address taken and is not referenced anywhere else in the loop, including all loops that are nested. For example, in the following code, only **S** in the nested loop is recognized as a reduction:

```
int i,j, S=0;
#pragma ibm parallel_loop
for (i= 0 ;i < N; i++) {
    S = S+ i;
    #pragma ibm parallel_loop
    for (j=0;j< M; j++) {
        S = S + j;
    }
}
```

op	Is one of the following operators: `+ - * ^	&`
assign_op	Is one of the following operators: `+= -= *= ^= |= &=`
expr	Is any valid expression.

Recognized reductions are listed by the **-qinfo=reduction** option. When using IBM directives, use critical sections to synchronize access to all reduction variables not recognized by the compiler. OpenMP directives provide you with mechanisms to specify reduction variables explictily.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Countable Loops (C Only)" on page 54
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"#pragma ibm critical Preprocessor Directive (C Only)" on page 245
"info" on page 144 Compiler Option

Shared and Private Variables in a Parallel Environment (C Only)

Variables can have either shared or private context in a parallel environment.

- Variables in shared context are visible to all threads running in associated parallel loops.
- Variables in private context are hidden from other threads. Each thread has its own private copy of the variable, and modifications made by a thread to its copy are not visible to other threads.

The default context of a variable is determined by the following rules:

- Variables with **static** storage duration are shared.
- Dynamically allocated objects are shared.
- Variables with automatic storage duration are private.
- Variables in heap allocated memory are shared. There can be only one shared heap.
- All variables defined outside a parallel construct become shared when the parallel loop is encountered.
- Loop iteration variables are private within their loops. The value of the iteration variable after the loop is the same as if the loop were run sequentially.
- Memory allocated within a parallel loop by the **alloca** function persists only for the duration of one iteration of that loop, and is private for each thread.

The following code segments show examples of these default rules:

```
int E1;                         /* shared static     */

 void main (argvc,...) {         /* argvc is shared   */
   int i;                        /* shared automatic  */

   void *p = malloc(...);        /* memory allocated by malloc   */
                                 /* is accessible by all threads */
                                 /* and cannot be privatized     */

   #pragma omp parallel firstprivate (p)
   {
     int b;                      /* private automatic  */
     static int s;               /* shared static      */
     #pragma omp for
     for (i =0;...) {
       = b;                      /* b is still private here !    */
       foo (i);                  /* i is private here because it */
                                 /* is an iteration variable     */
     }
     #pragma omp parallel
     {
       = b                       /* b is shared here because it  */
                                 /* is another parallel region   */
     }
   }
 }
```

```
int E2;                         /*shared static */

 void foo (int x) {              /* x is private for the parallel */
                                 /* region it was called from     */

   int c;                        /* the same */
 ... }
```

The compiler can privatize some shared variables if it is possible to do so without changing the semantics of the program. For example, if each loop iteration uses a unique value of a shared variable, that variable can be privatized. Privatized shared variables are reported by the **-qinfo=private** option. Use critical sections to synchronize access to all shared variables not listed in this report.

Some OpenMP preprocessor directives let you specify visibility context for selected data variables. For more information, see OpenMP directive descriptions or the OpenMP C and C++ Application Program Interface specification.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Countable Loops (C Only)" on page 54
"Reduction Operations in Parallelized Loops (C Only)" on page 56

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"#pragma ibm critical Preprocessor Directive (C Only)" on page 245
"info" on page 144 Compiler Option
OpenMP Specifications

Control Parallel Processing with Pragmas (C Only)

Parallel processing operations are controlled by pragma directives in your program source. You can use either IBM or OpenMP parallel processng directives. Each have their own usage characteristics.

IBM Directives	**OpenMP Directives**
Syntax: `#pragma ibm` *`pragma_name_and_args`* *`<countable for|while|do loop>`* Pragma directives must appear immediately before the section of code to which they apply. For most parallel processing pragma directives this section of code must be a countable loop, and the compiler will report an error if one is not found. More than one parallel processing pragma directive can be applied to a countable loop. For example: `#pragma ibm independent_loop#pragma ibm independent_calls` `#pragma ibm schedule(static,5)` `<countable for|while|do loop>` Some pragma directives are mutually-exclusive of each other. If mutually-exclusive pragmas are specified for the same loop, the pragma last specified applies to the loop. In the example below, the **parallel_loop** pragma directive is applied to the loop, and the **sequential_loop** pragma directive is ignored. `#pragma ibm sequential_loop#pragma ibm parallel_loop` Other pragmas, if specified repeatedly for a given loop, have an additive effect. For example: `#pragma ibm permutation (a,b)` `#pragma ibm permutation (c)` is equivalent to: `#pragma ibm permutation (a,b,c)`	**Syntax:** `#pragma omp` *`pragma_name_and_args`* *`statement_block`* Pragma directives generally appear immediately before the section of code to which they apply. The **omp parallel** directive is used to define the region of program code to be parallelized. Other OpenMP directives define visibility of data variables in the defined parallel region and how work within that region is shared and synchronized. For example, the following example defines a parallel region in which iterations of a **for** loop can run in parallel: `#pragma omp parallel { #pragma omp for for (i=0; i<n; i++) ... }` This example defines a parallel region in which two or more non-iterative sections of program code can run in parallel: `#pragma omp sections { #pragma omp section structured_block_1 ... #pragma omp section structured_block_2 ... .... }`

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"smp (C Only)" on page 215 Compiler Option
"info" on page 144 Compiler Option

#pragma Preprocessor Directives for Parallel Processing (C Only)

The #pragma directives on this page give you control over how the compiler handles parallel processing in your program. These pragmas fall into two groups; IBM C for AIX-specific directives, and directives conforming to the OpenMP Application Program Interface specification.

Use the **-qsmp** compiler option to specify how you want parallel processing handled in your program. You can also instruct the compiler to ignore all parallel processing-related #pragma directives by specifying the **-qignprag=ibm:omp** compiler option.

Directives apply only to the statement or statement block immediately following the directive.

IBM Pragma Directives	Description
"#pragma ibm critical Preprocessor Directive (C Only)" on page 245	Instructs the compiler that the statement or statement block immediately following this pragma is a critical section.
"#pragma ibm independent_calls Preprocessor Directive (C Only)" on page 245	Asserts that specified function calls within the chosen loop have no loop-carried dependencies.
"#pragma ibm independent_loop Preprocessor Directive (C Only)" on page 246	Asserts that iterations of the chosen loop are independent, and that the loop can therefore be parallelized.
"#pragma ibm iterations Preprocessor Directive (C Only)" on page 246	Specifies the approximate number of loop iterations for the chosen loop.
"#pragma ibm parallel_loop Preprocessor Directive (C Only)" on page 247	Explicitly instructs the compiler to parallelize the chosen loop.
"#pragma ibm permutation Preprocessor Directive (C Only)" on page 248	Asserts that specified arrays in the chosen loop contain no repeated values.
"#pragma ibm schedule Preprocessor Directive (C Only)" on page 248	Specifies scheduling algorithms for parallel loop execution.
"#pragma ibm sequential_loop Preprocessor Directive (C Only)" on page 250	Explicitly instructs the compiler to execute the chosen loop sequentially.

OpenMP Pragma Directives	Description
"#pragma omp parallel Preprocessor Directive (C Only)" on page 258	Defines a parallel region to be run by multiple threads in parallel. With specific exceptions, all other OpenMP directives work within parallelized regions defined by this directive.

“#pragma omp for Preprocessor Directive (C Only)” on page 253	Work-sharing construct identifying an iterative for-loop whose iterations should be run in parallel.
“#pragma omp parallel for Preprocessor Directive (C Only)” on page 260	Shortcut combination of **omp parallel** and **omp for** pragma directives, used to define a parallel region containing a single **for** directive.
“#pragma omp sections Preprocessor Directive (C Only)” on page 261	Work-sharing construct identifying a non-iterative section of code containing one or more subsections of code that should be run in parallel.
“#pragma omp parallel sections Preprocessor Directive (C Only)” on page 260	Shortcut combination of **omp parallel** and **omp sections** pragma directives, used to define a parallel region containing a single **sections** directive.
“#pragma omp single Preprocessor Directive (C Only)” on page 262	Work-sharing construct identifying a section of code that must be run by a single available thread.
“#pragma omp master Preprocessor Directive (C Only)” on page 257	Synchronization construct identifying a section of code that must be run only by the master thread.
“#pragma omp critical Preprocessor Directive (C only)” on page 252	Synchronization construct identifying a statement block that must be executed by a single thread at a time.
“#pragma omp barrier Preprocessor Directive (C Only)” on page 252	Synchronizes all the threads in a parallel region.
“#pragma omp flush Preprocessor Directive (C Only)” on page 253	Synchronization construct identifying a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.

RELATED CONCEPTS

“Chapter 2. Program Parallelization (C Only)” on page 53

RELATED TASKS

“Control Parallel Processing with Pragmas (C Only)” on page 59

RELATED REFERENCES

List of Preprocessor Directives
“IBM Run-time Options for Parallel Processing (C Only)” on page 63
“OpenMP Run-time Options for Parallel Processing (C Only)” on page 66
“Built-in Functions Used for Parallel Processing (C Only)”
“smp (C Only)” on page 215 compiler option
OpenMP Specifications

Built-in Functions Used for Parallel Processing (C Only)

Use these built-in functions to obtain information about the parallel environment.Function definitions for the **omp_** functions can be found in the **omp.h** header file.

Function Prototype	Description
`int __parthds(void)`	This function returns the value of the **parthds** run-time option. If the **parthds** option is not explicitly set by the user, the function returns the default value set by the run-time library. If the **-qsmp** compiler option was not specified during program compilation, this function returns 1 regardless of run-time options selected.
`int __usrthds(void)`	This function returns the value of the **usrthds** run-time option. If the **usrthds** option is not explicitly set by the user, or the **-qsmp** compiler option was not specified during program compilation, this function returns 0 regardless of run-time options selected.
`int omp_get_num_threads(void);`	This function returns the number of threads currently in the team executing the parallel region from which it is called.
`int omp_get_max_threads(void);`	This function returns the maximum value that can be returned by calls to omp_get_num_threads.
`int omp_get_thread_num(void);`	This function returns the thread number, within its team, of the thread executing the function. The thread number lies between 0 and omp_get_num_threads()-1, inclusive. The master thread of the team is thread 0.
`int omp_get_num_procs(void);`	This function returns the maximum number of processors that could be assigned to the program.
`int omp_in_parallel(void);`	This function returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0.
`void omp_set_dynamic(int dynamic_threads);`	This function enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
`int omp_get_dynamic(void);`	This function returns non-zero if dynamic thread adjustments enabled and returns 0 otherwise.
`void omp_set_nested(int nested);`	This function enables or disables nested parallelism.
`int omp_get_nested(void);`	This function returns non-zero if nested parallelism is enabled and 0 if it is disabled.

`void omp_init_lock(omp_lock_t *lock);`
`void omp_init_nest_lock(omp_nest_lock_t *lock);`
	These functions provide the only means of initializing a lock. Each function initializes the lock associated with the parameter lock for use in subsequent calls.

`void omp_destroy_lock(omp_lock_t *lock);`
`void omp_destroy_nest_lock(omp_nest_lock_t *lock);`
	These functions ensure that the pointed to lock variable lock is uninitialized.

`void omp_set_lock(omp_lock_t *lock);`
`void omp_set_nest_lock(omp_nest_lock_t *lock);`
	Each of these functions blocks the thread executing the function until the specified lock is available and then sets the lock. A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function.

Function Prototype	Description
`void omp_unset_lock(omp_lock_t *lock);` `void omp_unset_nest_lock(omp_nest_lock_t *lock);`	These functions provide the means of releasing ownership of a lock.
`int omp_test_lock(omp_lock_t *lock);` `int omp_test_nest_lock(omp_nest_lock_t *lock);`	These functions attempt to set a lock but do not block execution of the thread.

Note:
In the current implementation, nested parallel regions are always serialized. As a result, **omp_set_nested** does not have any effect, and **omp_get_nested** always returns 0.

For complete information about OpenMP runtime library functions, refer to the OpenMP C/C++ Application Program Interface specification.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"IBM Run-time Options for Parallel Processing (C Only)"
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66
"smp (C Only)" on page 215 Compiler Option
OpenMP Specifications

IBM Run-time Options for Parallel Processing (C Only)

Run-time time options affecting parallel processing are specified in the XLSMPOPTS environment variable. This environment variable, which must be set before you run an application, uses syntax of form:

```
XLSMPOPTS=option_and_args[:option_and_args][ ... ]
```

Parallelization run-time options can also be specified using OMP environment variables. When OMP and XLSMPOPTS run-time options conflict, OMP options will prevail.

Run-time options fall into different categories as described below.

Scheduling Algorithm Options

`schedule=`*`algorith`*`=[`*`n`*`]`	This option specifies the scheduling algorithm used for loops not explictly assigned a scheduling alogorithm with the **ibm schedule** pragma. Valid options for *algorithm* are: • guided • affinity • dynamic • static If specified, the value of *n* must be an integer value of 1 or greater. The default is scheduling algorithm is **static**. See "#pragma ibm schedule Preprocessor Directive (C Only)" on page 248 for a description of these algorithms.
Parallel Environment Options	
`parthds=`*`num`*	*num* represents the number of parallel threads requested, which is usually equivalent to the number of processors available on the system. Some applications cannot use more threads than the maximum number of processors available. Other applications can experience significant performance improvements if they use more threads than there are processors. This option gives you full control over the number of user threads used to run your program. The default value for *num* is the number of processors available on the system.
`usrthds=`*`num`*	*num* represents the number of user threads expected. This option should be used if the program code explicitly creates threads, in which case *num* should be set to the number of threads created. The default value for *num* is 0.
`stack=`*`num`*	*num* specifies the largest amount of space required for a thread's stack. The default value for *num* is 32768.
Performance Tuning Options	

`spins=`*`num`*	*num* represents the number of loop spins before a yield occurs. When a thread completes its work, the thread continues executing in a tight loop looking for new work. One complete scan of the work queue is done during each busy-wait state. An extended busy-wait state can make a particular application highly responsive, but can also harm the overall responsiveness of the system unless the thread is given instructions to periodically scan for and yield to requests from other applications. A complete busy-wait state for benchmarking purposes can be forced by setting both **spins** and **yields** to 0. The default value for *num* is 100.
`yields=`*`num`*	*num* represents the number of yields before a sleep occurs. When a thread sleeps, it completely suspends execution until another thread signals that there is work to do. This provides better system utilization, but also adds extra system overhead for the application. The default value for *num* is 100.
`delays=`*`num`*	*num* represents a period of do-nothing delay time between each scan of the work queue. Each unit of delay is achieved by running a single no-memory-access delay loop. The default value for *num* is 500.

Dynamic Profiling Options

profilefreq `=`*`num`*	*num* represents the sampling rate at which each loop is revisited to determine appropriateness for parallel processing. The run-time library uses dynamic profiling to dynamically tune the performance of automatically-parallelized loops. Dynamic profiling gathers information about loop running times to determine if the loop should be run sequentially or in parallel the next time through. Threshold running times are set by the **parthreshold** and **seqthreshold** dynamic profiling options, described below. If *num* is 0, all profiling is turned off, and overheads that occur because of profiling will not occur. If *num* is greater than 0, running time of the loop is monitored once every *num* times through the loop. The default for *num* is 16. The maximum sampling rate is 32. Higher values of *num* are changed to 32.

`parthreshold=`*`mSec`*	*mSec* specifies the expected running time in milliseconds below which a loop must be run sequentially. *mSec* can be specified using decimal places. If **parthreshold** is set to 0, a parallelized loop will never be serialized by the dynamic profiler. The default value for *mSec* is 0.2 milliseconds.
`seqthreshold=`*`mSec`*	*mSec* specifies the expected running time in milliseconds beyond which a loop that has been serialized by the dynamic profiler must revert to being run in parallel mode again. *mSec* can be specified using decimal places. The default value for *mSec* is 5 milliseconds.

Note: You must use thread-safe compiler mode invocations when compiling parallelized program code.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54
"Compiler Modes" on page 1

RELATED TASKS

"Invoke the Batch Compiler" on page 15

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"Built-in Functions Used for Parallel Processing (C Only)" on page 61
"OpenMP Run-time Options for Parallel Processing (C Only)"
"smp (C Only)" on page 215 Compiler Option

OpenMP Run-time Options for Parallel Processing (C Only)

OpenMP run-time time options affecting parallel processing are specified in a set of OMP environment variables. These environment variables, which must be set before you run an application, use syntax of form:

```
    env_variable=option_and_args
```

Parallelization run-time options can also be specified by the XLSMPOPTS environment variable. When OMP and XLSMPOPTS run-time options conflict, OMP options will prevail.

OpenMP run-time options fall into different categories as described below.

Scheduling Algorithm Environment Variable

`OMP_SCHEDULE=`*`algorithm`*	This option specifies the scheduling algorithm used for loops not explictly assigned a scheduling alogorithm with the **omp schedule** directive. For example: `OMP_SCHEDULE="guided, 4"` Valid options for *algorithm* are: • dynamic[, *n*] • guided[, *n*] • runtime • static[, *n*] If specified, the value of *n* must be an integer value of 1 or greater. The default is scheduling algorithm is **static**. See #pragma omp for Preprocessor Directive for a description of these algorithms.

Parallel Environment Environment Variables

`OMP_NUM_THREADS=`*`num`*	*num* represents the number of parallel threads requested, which is usually equivalent to the number of processors available on the system. This number can be overridden during program execution by calling the **omp_set_num_threads( )** runtime library function. Some applications cannot use more threads than the maximum number of processors available. Other applications can experience significant performance improvements if they use more threads than there are processors. This option gives you full control over the number of user threads used to run your program. The default value for *num* is the number of processors available on the system.

`OMP_NESTED=TRUE|FALSE`	This environment variable enables or disables nested parallelism. The setting of this environment variable can be overrridden by calling the **omp_set_nested( )** runtime library function. If nested parallelism is disabled, nested parallel regions are serialized and run in the current thread. In the current implementation, nested parallel regions are always serialized. As a result, OMP_SET_NESTED does not have any effect, and **omp_get_nested()** always returns 0. If **-qsmp=nested_par** option is on (only in non-strict OMP mode), nested parallel regions may employ additional threads as available. However, no new team will be created to run nested parallel regions. The default value for OMP_NESTED is FALSE.
Dynamic Profiling Environment Variable	
`OMP_DYNAMIC=TRUE|FALSE`	This environment variable enables or disables dynamic adjustment of the number of threads available for running parallel regions. If set to TRUE, the number of threads available for executing parallel regions may be adjusted at runtime to make the best use of system resources. See the description for profilefreq=*num* in IBM Run-time Options for Parallel Processing for more information. If set to FALSE, dynamic adjustment is disabled. The default setting is TRUE.
Note:	You must use thread-safe compiler mode invocations when compiling parallelized program code.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Compiler Modes" on page 1

RELATED TASKS

"Invoke the Batch Compiler" on page 15

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"IBM Run-time Options for Parallel Processing (C Only)" on page 63
"#pragma ibm schedule Preprocessor Directive (C Only)" on page 248
"Built-in Functions Used for Parallel Processing (C Only)" on page 61
"smp (C Only)" on page 215

Chapter 3. Shared Libraries

Constructing a Library

Libraries with Non-Shared Files
Non-shared files are files that are linked into the C++ executable program.

If you are using the incremental compiler, you can construct a library containing non-shared files by ensuring that your configuration file includes a directive such as the following:

```
 target "bar.o" { // non-shared object
```

source “example.C”

}

run after targets(“bar.o”) “ar rv libfoo.a foo.o bar.o”

If you are using the batch compiler, construct a library using these steps:
- Compile each file using the **-qpriority=***N* compiler option and if applicable in your application, **#pragma priority**(*N*) directives within the file. Normally, simply specifying the priority level for the file with the -qpriority=N option is sufficient. Use numbers for *N* to specify the priority levels at which you want objects initialized.
- Use the AIX **ar** command to link the files and produce an archive library file.

```
xlC -c -o bar.o example.C
ar rv libfoo.a bar.o
```

Libraries with Shared Files
Shared files are files that are used by more than one program.

If you are using the incremental compiler, you can construct a library containing shared files by ensuring that your configuration file includes a directive such as the following:

target type (shr) “foo.o” { // shared object

source “example.C”

}

run after targets(“bar.o”) “ar rv libfoo.a foo.o bar.o”

You can combine several of these libraries using the AIX **ar** command. The ar command is used to collect object files into one file (an archive file).

If you are using the traditional batch compiler you should compile with the **-qmkshrobj** compiler option or use the **makeC++SharedLib** program found in /usr/vacpp/bin and the AIX **ar** command.

```
xlC -c example.C
xlC -qmkshrobj -o foo.o
ar rv libfoo.a foo.o bar.o
```

RELATED TASKS

"Initialize Shared Library (C++)"
"Specify Priority Levels for Library Objects (C++, AIX)" on page 72
"Example of Object Initialization in a Group of Files (C++)" on page 71
Produce Object Files from a Build

RELATED REFERENCES

ar Command
"makeC++SharedLib Command" on page 79
"mkshrobj" on page 184

Initialize Shared Library (C++)

In some C++ programs, it is important to specify the order in which objects are initialized.

Before the main function of a C++ program is executed, the language definition ensures that all objects with constructors from all the files included in the C++ program have been properly constructed. The language definition, however, does not specify the order of initialization for these objects across files. In some cases, you may want to specify the initialization order of some objects in your program.

Often, your program will be made up of several files and files contained in libraries. The libraries that you use with your C++ source program may contain object (.o) files that have components shared with other programs (shared files), as well as files that are only used by your program (non-shared files).

To specify an initialization order, you can:
- Specify an initialization priority number for objects within a file using the #pragma priority directive.
- Generate shared objects using the sharedLibPriority incremental build option (or the "mkshrobj" on page 184 batch compiler option), then construct an archive (.a) library containing several shared and non-shared objects.

The run-time environment initializes the objects in these libraries in the order of their priority number. Priority numbers can range from -2147483648 to 2147483647. However, numbers from -2147483648 to -2147482624 are reserved for system use. The highest priority you can specify is -2147482623, and it is initialized first. The lowest priority, 2147483647, is initialized last.

Objects in files with identical priorities are initialized in random order.

When your program exits, the destructors for global and static objects are invoked in the reverse order of their construction.

If you do not specify priority levels, the default priority is 0 (zero). If there are multiple shared objects with different priority levels, the priority levels determine the order in which they will be initialized. Within a single shared object or the main function, #pragma priority controls the initialization order. The executable program has a priority of 0.

The loadAndInit routine will initialize shared libraries. Likewise, terminateAndUnload will terminate them. Include the file load.h to use these routines, which are found in the libC.a library.

RELATED CONCEPTS

"Constructing a Library" on page 69

RELATED TASKS

"Specify Priority Levels for Library Objects (C++, AIX)" on page 72
"Example of Object Initialization in a Group of Files (C++)"

RELATED REFERENCES

ar Command
"makeC++SharedLib Command" on page 79
"mkshrobj" on page 184

Example of Object Initialization in a Group of Files (C++)

You can specify different priority numbers for objects within files, and the compiler will initialize them in the following order:

1. By reference (Incremental Builds Only)
   Reference order is determined by the incremental compiler according to the rules for orderless programming. For example, in the following case, since the constructor for A references b, b will be initialized first, even though it appears after a in the source file:
   extern B b;

       A a(b);

       B b;

2. By #pragma priority
3. By line and column within a file

The following example describes describes the initialization order for objects in two files, farm.C and zoo.C. Both files use #pragma priority directives. The following table shows part of the files with #pragma priority directives and hypothetical objects:

farm.C	zoo.C
`#pragma priority(20)`	`...`
`...`	`class lion K ;`
`class dog A ;`	`#pragma priority(30)`
`class dog B ;`	`class bear M ;`
`...`	`...`
`#pragma priority(100)`	`#pragma priority(50)`
`...`	`...`
`class cat C ;`	`class zebra N ;`
`class cow D ;`	`class snake S ;`
`...`	`...`
`#pragma priority(200)`	`#pragma priority(250)`
`class mouse E ;`	`class frog F ;`
`...`	`...`

The following configuration file will build farm.C and zoo.C with opt(basePriority,10) to produce an executable file called animals:

```
option opt(basePriority, 10)
  {
   target "animals"
{
  source "farm.C", "zoo.C"
}
  }
```

Initialization takes place in the following order:

Object	Priority Value	Comment
“lion K”	10	Takes priority number of file “zoo.o” (10) (Initialized first)
“dog A”	20	Takes #PRAGMA PRIORITY(20) priority.
“dog B”	20	Follows “dog A”
“bear M”	30	Next priority number, specified by #PRAGMA PRIORITY(30)
“zebra N”	50	Next priority number from #PRAGMA PRIORITY(50)
“snake S”	50	Follows with same priority
“cat C”	100	Next priority number
“cow D”	100	Follows with same priority
“mouse E”	200	Next priority number
“frog F”	250	Next priority number (Initialized last).

RELATED TASKS

“Initialize Shared Library (C++)” on page 70

RELATED REFERENCES

basePriority Incremental Optimization Option
“priority” on page 202 Batch Compiler Option
“mkshrobj” on page 184 Batch Compiler Option

Specify Priority Levels for Library Objects (C++, AIX)

These examples are intended to show how you can specify priority levels for objects within a file, at the file level, and at the library level. However, in most applications it is not necessary to specify more than one or two priority levels.

Specifying Priority Levels within a File
To specify the order of initialization of objects within a file, use the #pragma priority directive. You can use any number of directives within the file, but the priority numbers must be in increasing order. That is, you cannot specify an object with a smaller priority number after you have specified one with a larger priority number.

The following example shows how to specify the priority for several objects within a source file.

```
...
#pragma priority(5) //Following objects constructed with priority 5
...
static struct base A ;
class house B ;
...
#pragma priority(10) //Following objects constructed with priority 10
...
class barn C ;
...
#pragma priority(2) // Error - priority number must be larger
// than preceding number (10)
...
#pragma priority(20) //Following objects constructed with priority 20
...
class garage D ;
...
```

Specifying the Priority Level of a File
To specify the priority level of a file, use the basePriority incremental build option or the "noprint" on page 189 batch compiler option. Use this option if you want all the objects in the file to have the same priority level, and you do not want to write #pragma priority(*N*) directives in the file.

For example, adding opt(basePriority, 20) to the target directive in your incremental configuration file as follows below or using the batch compiler option -qpriority=20, is equivalent to using #pragma priority(20):

```
option opt(basePriority,20) {

target "farm.o" {

    source "farm.c"

    }
```

If there are no #pragma priority directives within the file, all objects within the file have the priority specified with opt(basePriority).

If there are #pragma priority directives within the file, all objects found within the file up to the first #pragma priority directive are given the same priority number as specified for the file. The objects after a #pragma priority directive are given that priority number of *N* until the next #pragma priority directive is encountered.

Within the file, the first #pragma priority must have a higher priority number than the number used in the opt(basePriority) (or -qpriority) option and subsequent #pragma priority directives must have increasing numbers.

RELATED TASKS
"Initialize Shared Library (C++)" on page 70
"Example of Object Initialization in a Group of Files (C++)" on page 71

Chapter 4. Tools and Utilities

ar Command

The **ar** command maintains the indexed libraries used by the linkage editor. The **ar** command combines one or more named files into a single archive file written in ar archive format. When the **ar** command creates a library, it creates headers in a transportable format; when it creates or updates a library, it rebuilds the symbol table.

For a full description of the **ar** command, refer to the AIX Version 4 Commands Reference.

RELATED CONCEPTS

"Constructing a Library" on page 69

RELATED TASKS

"Initialize Shared Library (C++)" on page 70

RELATED REFERENCES

"makeC++SharedLib Command" on page 79
"mkshrobj" on page 184 Batch Compiler Option

c++filt Name Demangling Utility

Note:

This section applies to C++ programs only. C language programs do not use name mangling.

When VisualAge C++ compiles a C++ program, it encodes all function names and certain other identifiers to include type and scoping information. This encoding process is called mangling. The linker uses these mangled names to ensure type-safe linkage. These mangled names appear in the object files and final executable file.

Tools that use these files, the AIX dump utility for example, have only the mangled names and not the original source-code names, and present the mangled name in their output. This output is undesirable because the names are difficult to read.

Two utilities convert the mangled names to their original source code names:

c++filt	A filter that demangles (decodes) mangled names.
demangle.h	A class library that you can use to develop tools to manipulate mangled names.

Demangling Compiled C++ Names with c++filt

The c++filt utility converts mangled names to demangled names. You can enter any mangled name and obtain the demangled name. The filter copies characters from the given file names or standard input to standard output, replacing all mangled names with their corresponding demangled names.

To use the c++filt utility, type /usr/vacpp/bin/c++filt followed by any options, on the shell command line.

Syntax of the c++filt Utility

```
                                                                     +---------+
                                                                     V                 |
>>- c++filt -+----+-+----+-+----------+-+----+-+----+--+---------+-+->
             +- -m -+  +- -s -+  +- -w width -+  +- -C -+  +- -S -+   +- filename -+
```

You can select one or more of the following options:

-m	Produces a symbol map on standard output. This map contains a list of the mangled names and their corresponding demangled names. This output follows the filtered output.
-s	Produces a side-by-side demangling, with each mangled name encountered in the input stream replaced by a combination of the demangled name followed by the original mangled name.
-w *width*	Prints demangled names in fields width characters wide. If the name is shorter than width, it is padded on the right with blanks; if longer, it is truncated to width.
-C	Demangles stand-alone class names such as Q2_1X1Y.
-S	Demangles special compiler generated symbol names such as __vft1X.
filename	Is the name of the file containing the mangled names you want to demangle. You can specify more than one filename.

The following example shows the symbols contained in an object file functions.o, producing a side-by-side listing of the mangled and demangled names with a field width of 40 characters:

```
dump -tv functions.o | c++filt -s -w 40
```

Demangling Compiled C++ Names with the demangle Class Library

The demangle class library contains a small class hierarchy that client programs can use to demangle names and examine the resulting parts of the name. It also provides a C-language interface for use in C programs. Although it is a C++ library, it uses no external C++ features so you can link it directly to C programs. Demangle is included as part of libC.a, and is automatically linked, when required, if libC.a is linked.

You can write programs that use demangle.h to take a mangled name and return the demangled name, and the separate parts of the name. For example, given the mangled name of a member function, you can:

- Get the text of the entire demangled name
- Get the text of the function name
- Get the text of the qualifier list and each of its parts
- Get the text of the parameter list and each of its parts
- Ask questions about whether the function is const or volatile

Using demangle.h in C++ Programs
To demangle a name (represented as a character array), create a dynamic instance of Name class, providing the character string to the class's constructor. For example, if the compiler mangled X::f(int) to the mangled name f__1XFi, in order to demangle the name, enter:

```
char *rest;
Name *name = Demangle("f__1XFi", rest) ;
```

The demangler classifies names into four categories: functions, member functions, variables, and member variables. Once your program constructs an instance of class Name, your program can ask what kind of Name the instance is, using the Kind method of Name. Based on the kind of name returned, the program can ask for the text of the different parts of the name, or the text of the entire name.

For the mangled name f__1XFi example, you can find out the following:

```
name->Kind() == MemberFunction
((MemberFunctionName *)name)->Scope()->Text() is "X"
((MemberFunctionName *)name)->RootName() is "f"
((MemberFunctionName *)name)->Text() is "X::f(int)"
```

If the supplied character string is not a name that requires demangling, because the original name was not mangled, the Demangle function returns NULL.

For further details about the demangle.h library and the C++ interface, look at the comments in the library's header file, /usr/vacpp/include/demangle.h.

CreateExportList Command

Syntax
CreateExportList [-r] *exp-listname* [-f *filelistname* | *obj_files*] [-X 32|64]
where:
exp-listname is the file name that contains the list of global symbols found in the object files specified by either *obj_files* or -f *filelistname*.
obj_files is actual names of object files.
filelistname is the file that contains a list of object filenames.
Description
This command creates a file containing a list of all the global symbols found in a

given set of object files. Template prefixes are prunned, and `__rsrc` ignored if **-r** is specified. The **-f** option specifies a file containing the list of object files. The first argument is the export list file name (which is overwritten) and the remaining arguments are the object files.

CreateExportList creates an empty list if any of the following are true:

- no object files are given
- the -f option is missing
- the file specified by -f is empty.

Suboptions

-r	Do not add resource file symbol (__rsrc) to the exports list (but still export it).
-X 32|64	Generate names from 32-bit or 64-bit object files in input list. -X32 is the default.

RELATED CONCEPTS

"Constructing a Library" on page 69

RELATED TASKS

"Initialize Shared Library (C++)" on page 70

RELATED REFERENCES

"mkshrobj" on page 184
"makeC++SharedLib Command" on page 79

IPF/X Compiler and Viewer

The Information Processing Facility (IPF/X) is a tool that you can use to:

- create online information,
- specify how it will appear on the screen,
- link various parts of the information together, and,
- provide help information that can be requested by the user.

IPF/X features include:

- A tagging language that formats text and provides ways to connect information and customize the information display.
- A compiler that creates online documents and help windows.
- A viewing program that displays formatted online documents.

Information about using IPF/X can be found by entering the commands shown below on the AIX command line:

IPF/X Information Title	Command to View Information
User's Guide	/usr/bin/ipfxug
Programming Reference	/usr/bin/ipfxref

RELATED TASKS

Compile IPF Help During a Build

linkxlC Command

linkxlC is a small shell script that links C++ .o and .a files. It can be redistributed and used by someone who does not have VisualAge C++ installed. It runs the **munch** utility, and then invokes **ld**.

The script supports linker options. It ignores most other options.

For a full description of the **ld** command, refer to the AIX Version 4 Commands Reference.

RELATED REFERENCES

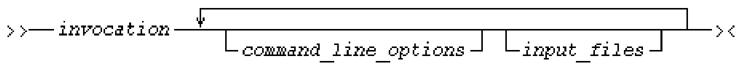
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Linkage Options" on page 42

makeC++SharedLib Command

makeC++SharedLib is a shell script that links C++ .o and .a files. It can be redistributed and used by someone who does not have VisualAge C++ installed.

We recommend that you use the -qmksrobj batch compiler option instead of the **makeC++SharedLib** command. The advantage to using this option is that the compiler will automatically include and compile the template instantiations in the tempinc directory.

This page describes the **makeC++SharedLib** command and provides an example of how to use it to make two shared libraries. It describes how to combine these two files in a library using the **ar** command so that object initialization takes place in the specified order.



Command Line Options

invocation	Is the path name for the **makeC++SharedLib** command that constructs the shared library file.
-o*shared_file*.o	Is the name of the file that will hold the shared file information. The default is shr.o.
-bOptions	Uses the **-b** binder options of the **ld** command.
-L*lib_dir*	Uses the **-L** option of the **ld** command to add the directory *lib_dir* to the list of directories to be searched for unresolved symbols. The **ld** command is described in the AIX Version 4 Commands Reference.
-l*library*	Adds *library* to the list of libraries to be searched for unresolved symbols.
-p *priority*	Specifies the priority level for the file. *priority* may be any number from -214782623 (highest priority-initialized first) to 214783647 (lowest priority-initialized last). Numbers from -214783648 to -214782624 are reserved for system use.

-I *import_list*	Uses the **-bI** option of the **ld** command to resolve the list of symbols in the file *import_list* that can be resolved by the binder.
-E *export_list*	Uses the **-bE** option of the **ld** command to export the external symbols in the export_list file. If you do not specify **-E** *export_list*, a list of all global symbols is generated.
-e *file*	Saves in file the list computed by **-E** *export_list*.
-n *name*	Sets the entry name for the shared executable to name. This is equivalent to using the command **ld -e** *name*
-X *mode*	Specifies the type of object file makeC++SharedLib should create. The mode must be either 32, which processes only 32-bit object files, or 64, which processes only 64-bit object files. The default is to process 32-bit object files (ignore 64-bit objects). The mode can also be set with the OBJECT_MODE environment variable. For example, OBJECT_MODE=64 causes makeC++SharedLib to process any 64-bit objects and ignore 32-bit objects. The -X flag overrides the OBJECT_MODE variable.

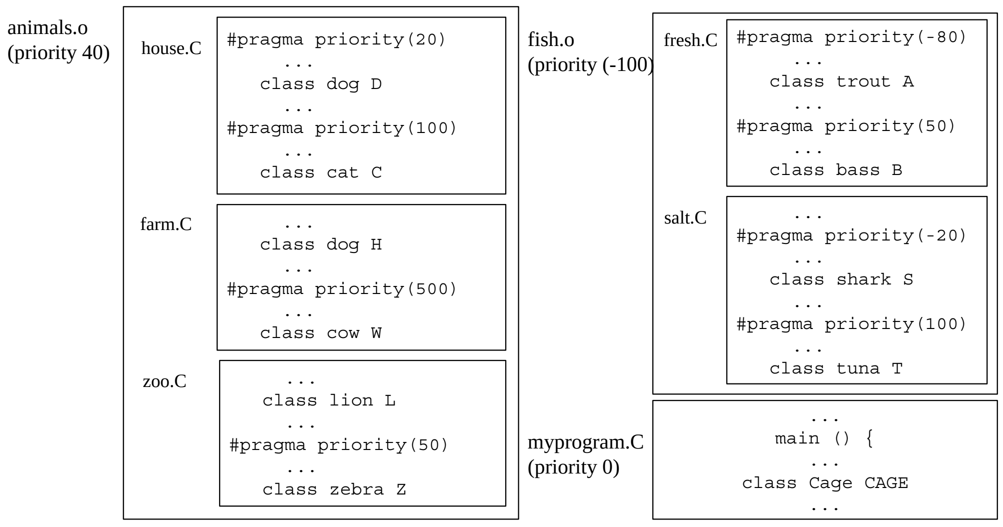
Input Files

file.o	Is an object file to be put into the shared library.
file.a	Is an archive file to be put into the shared library.

Example

The following example shows how to construct two shared libraries using the **makeC++SharedLib** command, and then use the AIX **ar** command to combine these libraries along with a file that contains the main function so that objects are initialized in the specified order.

The drawing below shows how the objects in this example are arranged in various files.



```
#pragma priority(20)
    ...
   class dog D
    ...
#pragma priority(100)
    ...
   class cat C
```

```
    ...
   class dog H
    ...
#pragma priority(500)
    ...
   class cow W
```

```
    ...
   class lion L
    ...
#pragma priority(50)
    ...
   class zebra Z
```

```
#pragma priority(-80)
    ...
   class trout A
    ...
#pragma priority(50)
    ...
   class bass B
```

```
    ...
#pragma priority(-20)
    ...
   class shark S
    ...
#pragma priority(100)
    ...
   class tuna T
```

```
      ...
   main () {
      ...
 class Cage CAGE
      ...
```

The first part of this example shows how to use **makeC++SharedLib** along with the **-qpriority=***N* option and the **#pragma priority**(*N*) directive to specify the initialization order for objects in these files.

The example shows how to make two shared libraries: animals.o containing object files compiled from house.C, farm.C, and zoo.C, and fish.o containing object files compiled from fresh.C and salt.C.

The example shows how to specify priorities and use the ar command so that all the objects in fish.o are initialized before the objects in myprogram.o, and all the objects in animals.o are initialized after the objects in myprogram.o. Within animals.o, the objects in zoo.C are initialized before the objects in house.C and farm.C.

To specify this initialization order, follow these steps:

1. Develop an initialization order for the objects in house.C, farm.C, and zoo.C:
   a. To ensure that the object lion L in zoo.C is initialized before any other objects in either of the other two files, compile zoo.C using a **-qpriority=***N* option with *N* less than zero so both objects have a priority number less than any other objects in farm.C and house.C:
      ```
      xlC zoo.C -c -qpriority=-50
      ```
   b. Compile the house.C and farm.C files without specifying the **-qpriority=***N* option (so *N*=0) so objects within the files retain the priority numbers specified by their **#pragma priority**(*N*) directives:
      ```
      xlC house.C farm.C -c
      ```
   c. Combine these three files in a shared library. Use **makeC++SharedLib** to construct a library animals.o with a priority of 40:
      ```
      makeC++SharedLib -o animals.o -p 40 house.o farm.o zoo.o
      ```
2. Develop an initialization order for the objects in fresh.C, and salt.C:
   a. Compile the fresh.C and salt.C files:
      ```
      xlC fresh.C salt.C -c
      ```
   b. To assure that all objects in fresh.C and salt.C are initialized before any other objects, use **makeC++SharedLib** to construct a library fish.o with a priority of -100.
      ```
      makeC++SharedLib -o fish.o -p -100 fresh.o salt.o
      ```

Because the shared library fish.o has a lower priority number (-100) than animals.o (40), when the files are placed in an archive file with the **ar** command, their objects are initialized first.

3. Compile myprogram.C that contains the function main to produce an object file myprogram.o. By not specifying a priority, this file is compiled with a default priority of zero, and the objects in main have a priority of zero.

   ```
   xlC myprogram.C -c
   ```

4. To create a library that contains the two shared libraries, and the program myprogram.o that contains the function main, so that the objects are initialized in the order you have specified, you use the **ar** command. To produce an archive file, prio_lib.a, enter the command:

   ```
   ar rv prio_lib.a animals.o fish.o myprogram.o
   ```

   where:

rv	Are two **ar** options. r replaces a named file if it already appears in the library, and v writes to standard output a file-by-file description of the making of the new library.
prio_lib.a	Is the name you specified for the archive file that will contain the shared library files and their priority levels.
animals.o fish.o	Are the two shared files you created with **makeC++SharedLib**.
myprogram.o	Is the name of the file that contains the function main.

The order of initialization of the objects is shown in the following table.

Order of Initialization of Objects in priolib.a			
File	**Class Object**	**Priority Value**	**Comment**
“fish.o”		-100	All objects in “fish.o” are initialized first because they are in a library prepared with **makeC++SharedLib** -p -100 (lowest priority number, -p -100, specified for any files in this compilation)
	“shark S”	-100(-200)	Initialized first in “fish.o” because within file, #pragma priority(-200)
	“trout A”	-100(-80)	#pragma priority(-80)
	“tuna T”	-100(10)	#pragma priority(10)
	“bass B”	-100(500)	#pragma priority(500)

Order of Initialization of Objects in priolib.a			
“myprog.o”		0	File generated with no priority specifications; default is 0
	“CAGE”	0(0)	Object generated in **main** with no priority specifications; default is 0
“animals.o”		40	File generated with **makeC++SharedLib** with -p 40
	“lion L”	40(-50)	Initialized first in file “animals.o” compiled with -qpriority=-50
	“horse H”	40(0)	Follows with priority of 0 (since -qpriority=*N* not specified at compilation and no #pragma priority(*N*) directive)
	“dog D”	40(20)	Next priority number (specified by #pragma priority(20))
	“zebra N”	40(50)	Next priority number from #pragma priority(50)
	“cat C”	40(100)	Next priority number from #pragma priority(100)
	“cow W”	40(500)	Next priority number from #pragma priority(500) (Initialized last)

5. To produce an executable file, animal_time, so that the objects are initialized in the order you have specified, enter:
   ```
   xlC prio_lib.a -oanimal_time
   ```
   You can place both nonshared and shared files with different priority levels in the same archive library using the AIX **ar** command.

RELATED CONCEPTS

“Constructing a Library” on page 69

RELATED TASKS

“Initialize Shared Library (C++)” on page 70

RELATED REFERENCES

ar Command
“mkshrobj” on page 184

Resource Compiler - An Overview

The *Resource Compiler* (**irc**) is an application-development tool that lets you add application resources such as message strings, pointers, and menus to the executable file of your application.

The Resource Compiler lets you quickly define or modify application resources without recompiling the application itself. This is especially important for international applications because it lets you define all language-dependent data, such as message strings, as resources. Preparing the application for a new language is simply a matter of relinking the new resources to an existing executable file.

RELATED REFERENCES

Resource Compiler - Resource Script Files
"Resource Compiler - Syntax"

Resource Compiler - Syntax

Syntax

The **irc** command line entry has the following syntax:

```
irc [options] resource_script_file [resource_file]
```

where:

resource_script_file	The filename of the resource script file to be compiled. If the file is not in the current directory, you must provide a full path. If you provide a filename without specifying a filename extension, **irc** automatically appends the **.rc** extension to the name.
resource_file	The name of the binary resource file to be created. If *resource_file* is not specified, the base name of the resource file defaults to the base name of the input resource script file. If the binary resource file does not already exist, **irc** creates it; otherwise, **irc** replaces the existing file. If the file is not in the current directory, you must provide a full path. The binary resource file must have the **.res** filename extension.

Options
The following options can be specified on the Resource Compiler command line:

Option	Description
`-D`*`defname[=value]`*	Define macro to preprocessor. This option is useful for passing conditional compilation flags to the preprocessor. *defname* is any sequence of letters, underscore symbols, and digits which does not begin with a digit. *value* is a sequence of symbols which you want to substitute for the *defname* wherever it appears in the input script file. If you omit *=value*, the value of *defname* will default to 1. For example, the option -D_3d is equivalent to including the following at the beginning of the input file: `    #define _3d 1` You can use the *-D* option up to 8 times to define different macros from the command line.

`-h`
`-?`
	Access Help
	
	These options cause the Resource Compiler to display a help summary of available options. When you use the -h or -? options, the Resource Compiler does not read any input files.
	
	Entering **irc** on the command line with no operands also displays the help summary, shown below.
	`irc (IBM Resource Compiler) Version 5.00.000 Oct 6 1997`
	`Copyright (C) IBM Corp. 1997.`
	`- Licensed Materials - Program Property of IBM - All Ri`
	`US Government Users Restricted Rights - Use, duplicatio`
	`restricted by GSA ADP Schedule Contract with IBM Corp.`
	`irc [<options>] <.rc input file> [<.res output file>]`
	`Options: -D<defname> - Preprocessor define`
	`-I<path> - Include file path`
	`-lang language - language`
	`-n - Don't show logo`
	`-z - Ignore os2.h`
	`-w[1|2|4] - Suppress warnings`
	`-? - Access Help`
	`-h - Access Help`

`-I`*`path`*	Include file path. This option defines paths for files to be included with the source file. The *path* is any path where you want **irc** to search for files included by the preprocessor #include directive. The *path* must not contain embedded blanks. To include more than one path, code the -I option once for each path.
`-lang`*`language`*	Language The -lang option specifies the language in which the resource script is compiled. The value of *language* is a POSIX locale.

Option	Description

`-w`
`-w1`
`-w2`
`-w4`
	Suppress the display of all or selected levels of warning and informational messages.

`-z`	Ignore os.h header file. This option aids portability between AIX and OS/2® compilation environments. It causes the Resource Compiler to ignore the os.h header file which is valid in the OS/2 environment, but not available for AIX.

RELATED TASKS

Resource Compiler - Examples

RELATED REFERENCES

"Resource Compiler - An Overview" on page 84
Resource Compiler - Resource Script Files

Resource Compiler - Defining Constants

The -D option lets you define up to eight symbolic constants on the command line. The syntax is:

```
-Ddefname[=value]
```

where *defname* is a name, and *value* is an integer constant or an expression. The -D option is useful for passing conditional-compilation flags to the **irc** preprocessor.

The following example defines two symbolic constants to be passed to the preprocessor:

```
irc -DDEBUG -DVERSION=2 example
```

RELATED REFERENCES

"Resource Compiler - Syntax" on page 84

Resource Conversion Utility - Overview

The Resource Conversion Utility converts Windows-specific resource files for use with AIX. This is an applications development tool used in creating cross-platform applications.

Like source files, resource files are compiled with a resource compiler. Because resource files are unique to the environment that they have been developed in, resources cannot simply be inherited by similar applications developed under different environments. To create the same application and utilize the same resources for a different platform, the format of the resource must be converted to suit the operating environment. The resource conversion utility enables resources to be ported to other platforms without having to re-create the objects for similar applications.

Note:	The AIX version of VisualAge C++ can accept Windows® format icons, bitmaps, and cursors without conversion.

RELATED REFERENCES

"Resource Conversion Utility - Syntax"

Resource Conversion Utility - Syntax

Syntax

`irccnv [`*options*`] [`*infile*`] [`*outfile*`]`

where:

infile	Specifies the name of an AIX resource file. Do not specify *infile* if using the -i option, described below..	
outfile	Specifies the name of the output resource file. Do not specify *outfile* if using the -c or -e options, described below.	
options	`-c`	Output from **irccnv** will be redirected to standard out. Do not specify this option if you are providing a file name for *outfile*.
	`-e`	Output from **irccnv** will be redirected to standard error. Do not specify this option if you are providing a file name for *outfile*.
	`-i`	Input to **irccnv** is will come from standard input. Do not specify this option if you are providing a file name for *infile*.
	`-s`	Suppress warnings. Unrecognized input lines will be turned into comment lines.

You can get usage help by entering **irccnv** by itself on the command line. A help summary similar to that shown below will appear:

```
IBM RC Converter
Version 1.1
- Licensed Material - Program Property of IBM
(C) Copyright IBM Corp. 1997 - All Rights Reserved.
Converts Resource files between Windows RC files and AIX RC files
Usage : irccnv <options> | <input file> | <output file>
Options:
-c : Output to standard out
-e : Output to standard error
-i : Input from stdin
-s : Suppress warnings; comment lines with unrecognized keywords
```

Example

The following example is the command line syntax to convert the Windows resource file, win_res.rc, to an AIX resource file, my_res.rc:

```
irccnv win_res.rc my_res.rc
```

RELATED CONCEPTS
"Resource Conversion Utility - Overview" on page 86

Appendix A. Compiler Options

+

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-+
```

Purpose
Compiles any file, *filename.nnn*, as a C++ language file, where *nnn* is any suffix other than **.o**, **.a**, or **.s**.

Notes:
If you do not use the **-+** option, files must have a suffix of **.C** (uppercase C), **.cc**, **.cpp**, or **.cxx** to be compiled as a C++ file. If you compile files with suffix **.c** (lowercase c) without specifying **-+**, the files are compiled as a C language file.

Example
To compile the file myprogram.cplspls as a C++ source file, enter:

```
xlC -+ myprogram.cplspls
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

#

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-#
```

Purpose
Traces the compilation without invoking anything. This option previews the compilation steps specified on the command line. When the **xlC** command is issued with this option, it names the programs within the preprocessor, compiler, and linkage editor that would be invoked, and the options that would be specified to each program. The preprocessor, compiler, and linkage editor are not invoked.

Notes:
The **-#** option overrides the "v" on page 238 option. It displays the same information as **-v**, but does not invoke the compiler. Information is displayed to standard output.

Use this command to determine commands and files will be involved in a particular compilation. It avoids the overhead of compiling the source code and overwriting any existing files, such as **.lst** files.

Example
To preview the steps for the compilation of the source file myprogram.c, enter:

```
xlC myprogram.c -#
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

32, 64

Option Type	Default Value	#pragma options	C	C++
-q*option*	-	-	x	x

Syntax

```
-q32 | -q64
```

Purpose
Selects either 32- or 64-bit compiler mode.

Notes
The **-q32** and **-q64** options override the compiler mode set by the value of the OBJECT_MODE environment variable, if it exists. If the **-q32** and **-q64** options are not specified, and the OBJECT_MODE environment variable is not set, the compiler defaults to 32-bit output mode.

If the compiler is invoked in in 64-bit mode, the __64BIT__ preprocessor macro is defined.

Use **-q32** and **-q64** options, along with the "arch" on page 96 and "tune" on page 230 compiler options, to optimize the output of the compiler to the architecture on which that output will be used. Refer to the Acceptable Compiler Mode and Processor Architecture Combinations table for valid combinations of the **-q32**, **-q64**, **-qarch**, and **-qtune** compiler options.

Example
To specify that the executable program testing compiled from myprogram.c is to run on a computer with a 32-bit PowerPC architecture, enter:

```
xlC -o testing myprogram.c -q32 -qarch=ppc
```

Important Notes!

1. If you mix 32-and 64-bit compilation modes for different source files, your XCOFF objects will not bind. You must recompile completely to ensure that all objects are in the same mode.
2. Your link options must reflect the type of objects you are linking. If you compiled 64-bit objects, you must link these objects using 64-bit mode.

RELATED TASKS

"Specify Batch Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 21

RELATED REFERENCES

Acceptable Compiler Mode and Processor Architecture Combinations
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

aggrcopy

Option Type	Default Value	#pragma options	C	C++
-q*option*	*See notes.*	-	x	x

Syntax

```
-qaggrcopy=overlap | -qaggrcopy=nooverlap
```

Purpose
Enables destructive copy operations for structures and unions.

Notes
If the **-qaggrcopy=nooverlap** compiler option is enabled, the compiler assumes that the source and destination for structure and union assignments do not overlap. This assumption lets the compiler generate faster code.

Default Setting
The default setting of this option is **-qaggrcopy=nooverlap** when compiling to the ANSI, SAA and SAAL2 language levels.

The default setting of this option is **-qaggrcopy=overlap** when compiling to the EXTENDED and CLASSIC language levels.

Programs that do not comply to the ANSI C standard as it pertains to non-overlap of source and destination assignment may need to be compiled with the **-qaggrcopy=overlap** compiler option.

Example

```
xlc myprogram.c -qaggrcopy=nooverlap
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

alias

Option Type	Default Value	#pragma options	C	C++
-q*option*	ansi:typeptr:noallptrs:noaddrtaken*	ALIAS=*suboption*[*:suboption*]	x	x

Syntax

```
-qalias=suboption[:suboption][...]
ALIAS=suboption[:suboption]
```

Purpose
Requests the compiler to apply aliasing assertions to your compilation unit. The compiler will take advantage of the aliasing assertions to improve optimizations where possible, unless you specify otherwise.

Notes
If used, **#pragma ALIAS=***suboption* must appear before the first program statement.

The compiler will apply aliasing assertions according to the following *suboptions*:

`[NO]TYPeptr`	Pointers to different types are never aliased. In other words, in the compilation unit no two pointers of different types will point to the same storage location.
`[NO]ALLPtrs`	Pointers are never aliased (this also implies **-qalias=typeptr**). Therefore, in the compilation unit, no two pointers will point to the same storage location.
`[NO]ADDRtaken`	Variables are disjoint from pointers unless their address is taken. Any class of variable for which an address has *not* been recorded in the compilation unit will be considered disjoint from indirect access through pointers.
`[NO]ANSI`	Type-based aliasing is used during optimization, which restricts the lvalues that can be safely used to access a data object. The optimizer assumes that pointers can *only* point to an object of the same type. This (**ansi**) is the default for the **xlc**and **c89 compilers.**This option has no effect unless you also specify the "O, optimize" on page 192 option. If you select **noansi**, the optimizer makes worst case aliasing assumptions. It assumes that a pointer of a given type can point to an external object or any object whose address is already taken, regardless of type. This is the default for the **xlC** and **cc** compiler.

The following are not subject to type-based aliasing:
- Signed and unsigned types. For example, a pointer to a **signed int** can point to an **unsigned int**.
- Character pointer types can point to any type.
- Types qualified as **volatile** or **const**. For example, a pointer to a **const int** can point to an **int**.

Example
To specify worst-case aliasing assumptions when compiling myprogram.c, enter:

```
xlC myprogram.c -O -qalias=noansi
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
"Options that Specify Compiler Characteristics" on page 38
Equivalent Batch Compile-Link and Incremental Build Options

align

Option Type	Default Value	#pragma options	C	C++
-q*option*	align=full	ALIGN=*suboption*	x	x

Syntax

```
-qalign=suboption
ALIGN=suboption
```

Purpose
Specifies what aggregate alignment rules the compiler uses for file compilation. Use this option to specify the maximum alignment to be used when mapping a class-type object, either for the whole source program or for specific parts.

Notes
The **-qalign** suboptions are:

`power`	The compiler uses the RISC System/6000 alignment rules.
`full`	The compiler uses the RISC System/6000. alignment rules. The *power* option is the same as *full*.
`mac68k`	The compiler uses the Macintosh** alignment rules.
`twobyte`	The compiler uses the Macintosh alignment rules. The *mac68k* option is the same as *twobyte*.
`packed`	The compiler uses the **packed** alignment rules.
`bit_packed`	The compiler uses the **bit_packed** alignment rules. Alignment rules for **bit_packed** are the same as that for **packed** alignment except that bitfield data is packed on a bit-wise basis without respect to byte boundaries.
`natural`	The compiler maps structure members to their natural boundaries. This has the same effect as the *power* suboption, except that it also applies alignment rules to **doubles** and **long doubles** that are not the first member of a structure or union.

If you use the **qalign** option more than once on the command line, the last alignment rule specified applies to the file.

Within your source file, you can use **#pragma options align=reset** to revert to a previous alignment rule. The compiler stacks alignment directives, so you can go back to using the previous alignment directive, without knowing what it is, by specifying the **#pragma align=reset** directive. For example, you can use this option if you have a class declaration within an include file and you do not want the alignment rule specified for the class to apply to the file in which the class is included.

You can code **#pragma options align=reset** in a source file to change the alignment option to what it was before the last alignment option was specified. If no previous alignment rule appears in the file, the alignment rule specified in the invocation command is used.

Example 1 - Imbedded #pragmas
Using the compiler invocation:

```
xlC -qalign=mac68k file.c  /* <— default alignment rule for file is */
                           /*    Macintosh                          */
```

Where file.c has:

```
struct A {
  int a;
  struct B {
    char c;
    double d;
#pragma options align=power /* <— B will be unaffected by this      */
                            /*     #pragma, unlike previous behavior; */
                            /*     Macintosh alignment rules still    */
                            /*     in effect                          */
  } BB;
#pragma options align=reset /* <— A unaffected by this #pragma;      */
} AA;                       /*     Macintosh alignment rules still    */
                            /*     in effect                          */
```

Example 2 - Affecting Only Aggregate Definition
Using the compiler invocation:

```
xlC file2.c /* <— default alignment rule for file is                 */
            /*     RISC System/6000 since no alignment rule specified */
```

Where file2.c has:

```
extern struct A A1;
typedef struct A A2;
#pragma options align=packed /* <— use packed alignment rules        */
struct A {
  int a;
  char c;
};
#pragma options align=reset /* <— Go back to default alignment rules */
struct A A1;  /* <— aligned using packed alignment rules since      */
A2 A3;        /*     this rule applied when struct A was defined      */
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
__align Specifier
RISC System/6000 Alignment Rules
Macintosh and Twobyte Alignment Rules
Packed Alignment Rules
Alignment Rules for Nested Aggregates
Equivalent Batch Compile-Link and Incremental Build Options

alloc

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-qalloca
```

Purpose
Substitutes inline code for calls to function **alloca** as if **#pragma alloca** directives are in the source code.

Notes
If **#pragma alloca** is unspecified, or if you do not use **-ma**, **alloca** is treated as a user-defined identifier rather than as a built-in function.

C++ programs must specify **#include <malloc.h>** to include the **alloca** function declaration.

Example
To compile myprogram.c so that calls to the function **alloca** are treated as inline, enter:

```
xlC myprogram.c -qalloca
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

ansialias

Option Type	Default Value	#pragma options	C	C++
-q*option*	ansialias*	ANSIALIAS	x	x

Syntax

```
-qansialias | -qnoansialias
ANSIALIAS | NOANSIALIAS
```

Purpose
Specifies whether type-based aliasing is to be used during optimization. Type-based aliasing restricts the lvalues that can be used to access a data object safely.

Default Values
The default with **xlc**, **xlC** and **c89** is **ansialias**. The optimizer assumes that pointers can *only* point to an object of the same type.

The default with **cc** is **noansialias**.

Notes
This option is obsolete. Use "alias" on page 91 in your new applications.

This option has no effect unless you also specify the "O, optimize" on page 192 option.

If you select **noansialias**, the optimizer makes worst-case aliasing assumptions. It assumes that a pointer of a given type can point to an external object or any object whose address is already taken, regardless of type.

The following are not subject to type-based aliasing:
- Signed and unsigned types; for example, a pointer to a **signed int** can point to an **unsigned int**.
- Character pointer types can point to any type.
- Types qualified as **volatile** or **const**; for example, a pointer to a **const int** can point to an **int**.

Example
To specify worst-case aliasing assumptions when compiling myprogram.c, enter:

```
xlC myprogram.c -O -qnoansialias
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

arch

Option Type	Default Value	#pragma options	C	C++
-q*option*	arch=com	-	x	x

Syntax

```
-qarch=suboption
```

Purpose
Specifies the general processor architecture for which the code (instructions) should be generated.

Notes
If you want maximum performance on a specific architecture and will not be using the program on other architectures, use the appropriate architecture option. You can specify the architecture using the following basic suboptions:

`com`	In 32-bit execution mode, produces object code that contains instructions that will run on all the POWER, POWER2*, and PowerPC* hardware platforms (that is, the instructions generated are *common* to all platforms. Using **-qarch=com** is referred to as compiling in *common mode*. In 64-bit mode, produces object code that will run on all the 64-bit PowerPC hardware platforms but not 32-bit-only platforms. This is the default option unless the -O4 compiler option was specified. Defines the _ARCH_COM macro and produces portable programs.
`pwr`	Produces object code that contains instructions that will run on any of the POWER and POWER2 and 601 hardware platforms. Defines the _ARCH_PWR macro.
`pwr2`	Produces object code that contains instructions that will run on the POWER2 hardware platforms including pwr2s and p2sc. Defines the _ARCH_PWR and _ARCH_PWR2 macros.
`pwr3`	Produces object code that contains instructions that will run on the POWER3 hardware platforms. Defines the _ARCH_PWR and _ARCH_PWR3 macros.
`ppc`	In 32-bit mode, produces object code that contains instructions that will run on any of the 32-bit PowerPC hardware platforms. This suboption will cause the compiler to produce single-precision instructions to be used with single-precision data. In 64-bit mode, produces object code that will run on any of the 64-bit PowerPC hardware platforms. Defines the _ARCH_PPC macro.
`ppcgr`	In 32-bit mode, produces object code that contains optional graphics instructions for PowerPC processors. In 64-bit mode, produces object code that contains optional graphics instructions for 64-bit PowerPC hardware platforms. Defines the _ARCH_PPC and _ARCH_PPCGR macros.
`auto`	Produces object code that contains instructions that will run on the hardware platform on which it is compiled.

You can use **-qarch=***suboption* with **“tune” on page 230=***suboption*. **-qarch=***suboption* specifies the architecture for which the instructions are to be generated, and **-qtune=***suboption* specifies the target platform for which the code is optimized.

Default
The default setting of **-qarch** is **-qarch=com** unless the OBJECT_MODE environment variable is set to **64**..

Example
To specify that the executable program testing compiled from myprogram.c is to run on a computer with a 32-bit PowerPC architecture, enter:

```
    xlC -o testing myprogram.c -qarch=ppc
```

RELATED TASKS

"Specify Batch Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 21

RELATED REFERENCES

Acceptable Compiler Mode and Processor Architecture Combinations
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

assert

Option Type	Default Value	#pragma options	C	C++
-q*option*	noassert	-	x	

Syntax

```
-qassert=suboption
```

Purpose
Requests the compiler to apply aliasing assertions to your compilation unit. The compiler will take advantage of the aliasing assertions to improve optimizations where possible.

Notes
This option is obsolete. Use "alias" on page 91 in your new applications.

The compiler will apply aliasing assertions when you specify the following *suboptions*:

-qASSert=TYPeptr	Pointers to different types are never aliased. In other words, in the compilation unit no two pointers of different types will point to the same storage location.
-qASSert=ALLPtrs	Pointers are never aliased (this implies **-qassert=typeptr**). Therefore, in the compilation unit, no two pointers will point to the same storage location.
-qASSert=ADDRtaken	Variables are disjoint from pointers unless their address is taken. Any class of variable for which an address has *not* been recorded in the compilation unit will be considered disjoint from indirect access through pointers.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
Equivalent Batch Compile-Link and Incremental Build Options

attr

Option Type	Default Value	#pragma options	C	C++
-q*option*	noattr	ATTR	x	x

Syntax

```
-qattr | -qattr=full | -qnoattr
ATTR | ATTR=FULL | NOATTR
```

Purpose

Produces a compiler listing that includes an attribute listing for all identifiers.

Notes

`-qattr=full`	Reports all identifiers in the program.
`-qattr`	Reports only those identifiers that are used.

This option does not produce a cross-reference listing unless you also specify "xref" on page 241.

The "noprint" on page 189 option overrides this option.

If **-qattr** is specified after **-qattr=full**, it has no effect. The full listing is produced.

Example
To compile the program myprogram.c and produce a compiler listing of all identifiers, enter:

```
xlC myprogram.c -qxref -qattr=full
```

A typical cross-reference listing has the form:

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24 "Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

B

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

`-B | -B`*`prefix`*` | -B -t`*`programs`*` | -B`*`prefix`*` -t`*`programs`*

Purpose
Determines substitute path names for programs such as the compiler, assembler, linkage editor, and preprocessor.

Notes
The optional *prefix* defines part of a path name to the new programs. The compiler does not add a / between the
prefix and the program name.

To form the complete path name for each program, IBM VisualAge C++ adds prefix to the standard program names for the compiler, assembler, linkage editor and preprocessor.

Use this option if you want to keep multiple levels of some or all of IBM VisualAge C++ executables and have the option of specifying which one you want to use.

If **-B***prefix* is not specified, the default path is used.

-B"t" on page 227*programs* specifies the programs to which the **-B** prefix name is to be appended.

The **-B***prefix* "t" on page 227*programs* options override the "F" on page 124*config_file* option.

Example
To compile myprogram.c using a substitute **xlC** compiler in **/lib/tmp/mine/** enter:

```
xlC myprogram.c -B/lib/tmp/mine/
```

To compile myprogram.c using a substitute linkage editor in **/lib/tmp/mine/**, enter:

```
xlC myprogram.c -B/lib/tmp/mine/ -tl
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
Equivalent Batch Compile-Link and Incremental Build Options
"path" on page 197

bitfields

Option Type	Default Value	#pragma options	C	C++
-q*option*	unsigned	-	x	x

Syntax

```
-qbitfields=suboption
```

Purpose
Specifies if bitfields are signed. By default, bitfields are unsigned.

Notes
The **-qbitfields** suboptions are:

`signed`	Bitfields are signed.

`unsigned`	Bitfields are unsigned.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

bmaxdata

Option Type	Default Value	#pragma options	C	C++
-flag	-bmaxdata=0	-	x	x

Syntax

```
    -bmaxdata:number
```

This option sets the maximum size of the area shared by the static data (both initialized and uninitialized) and the heap to *size* bytes. This value is used by the system loader to set the soft ulimit.

The default setting is -bmaxdata=0.

Valid values of *number* are 0 and multiples of 0x10000000 (0x10000000, 0x20000000, 0x30000000, ...). The maximum value allowed by the system is 0x80000000.

If the value of *size* is 0, a single 256MB (0x10000000 byte) data segment (segment 2) will be shared by the static data, the heap, and the stack. If the value is non-zero, a data area of the specified size (starting in segment 3) will be shared by the static data and the heap, while a separate 256 MB data segment (segment 2) will be used by the stack. So, the total data size when 0 is specified 0 is 256MB, and the total size when 0x10000000 is specified is 512MB, with 256MB for the stack and 256MB for static data and the heap.

RELATED REFERENCES

Equivalent Batch Compile-Link and Incremental Build Options
"Options that Specify Linkage Options" on page 42

brtl

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
    -brtl
```

Purpose
Enables run-time linking for the output file. Run-time linking is the ability to resolve undefined and non-deferred symbols in shared modules after the program execution has already begun. It is a mechanism for providing run-time definitions (these function definitions are not available at link-time) and symbol

rebinding capabilities.
The *main* application must be built to enable run-time linking. You cannot simply link any module with the run-time linker.

Note
Do not specify this option if you are using the **xlC_r4** command because the DCE thread libraries are not compatible with runtime linking.

For more information on this and other -b linker options, see the AIX system documentation.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Linkage Options" on page 42
"bstatic, bdynamic, bshared" Compiler Option
Equivalent Batch Compile-Link and Incremental Build Options

bstatic, bdynamic, bshared

Option Type	Default Value	#pragma options	C	C++
-flag	bdynamic	-	x	x

Syntax

```
-bstatic | -bdynamic | -bshared
```

Purpose
Controls how shared objects are processed by the linkage editor.

Suboptions

`bdynamic,bshared`	Causes the linker to process subsequent shared objects in dynamic mode. This is the default. In dynamic mode, shared objects are not statically included in the output file. Instead, the shared objects are listed in the loader section of the output file.
`bstatic`	Causes the linker to process subsequent shared objects in static mode. In static mode, shared objects are statically linked in the output file.

The default option, **-bdynamic**, ensures that the C library (**lib.c**) links dynamically. To avoid possible problems with unresolved linker errors when linking the C library, you must add the **-bdynamic** option to the end of any compilation sections that use the **-bstatic** option.

For more information about this and other **ld** options, see your AIX system documentation.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Linkage Options" on page 42
Equivalent Batch Compile-Link and Incremental Build Options

C

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax
```
-C
```

Purpose
Preserves comments in preprocessed output.

Notes
The **-C** option has no effect without either the "E" on page 116 or the "P" on page 195 option. With the **-E** option, comments are written to standard output. With the **-P** option, comments are written to an output file.

Example
To compile myprogram.c to produce a file myprogram.i that contains the preprocessed program text including comments, enter:
```
xlC myprogram.c -P -C
```

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

c

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax
```
-c
```

Purpose
Instructs the compiler to pass source files to the compiler only.

Notes
The compiled source files are not sent to the linkage editor. The compiler creates an output object file, *file_name***.o**, for each valid source file, *file_name***.c** or *file_name***.i**.

The **-c** option is overridden if either the "E" on page 116, "P" on page 195, or "syntaxonly (C Only)" on page 225 options are specified.

The **-c** option can be used in combination with the "o" on page 190 option to provide an explicit name of the object file that is created by the compiler.

Example
To compile myprogram.c to produce an object file **myfile.o**, but no executable file, enter the command:
```
xlC myprogram.c -c
```

To compile myprogram.c to produce the object file **new.o** and no executable file, enter:

```
     xlC myprogram.c -c -o new.o
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

cache

Option Type	Default Value	#pragma options	C	C++
-q*option*			x	x

Syntax

```
-qcache=
{
        assoc=number |
        auto |
        cost=cycles |
        level=level |
        line=bytes |
        size=Kbytes |
        type=cache_type
}[: ...|
```

Purpose
The **-qcache** option specifies the cache configuration for a specific execution machine. If you know the type of execution system for a program, and that system has its instruction or data cache configured differently from the default case, use this option to specify the exact cache characteristics. The compiler uses this information to calculate the benefits of cache-related optimizations.

Notes
Allowable values for *suboption* are:

`assoc=`*`number`*	Specifies the set associativity of the cache. **0** Direct-mapped cache **1** Fully associate cache **N>1** n-way set associative cache
`auto`	Automatically detects the specific cache configuration of the compiling machine. This assumes that the execution environment will be the same as the compilation environment.
`cost=`*`cycles`*	Specifies the performance penalty resulting from a cache miss.

`level=`*`level`*	Specifies the level of cache affected. **1** Basic cache **2** Level-2 cache or, if there is no level-2 cache, the table lookaside buffer (TLB) **3** TLB If a machine has more than one level of cache, use a separate -qcache option.
`line=`*`bytes`*	Specifies the line size of the cache.
`size=`*`Kbytes`*	Specifies the total size of the cache.
`type=C|c|D|d|I|i`	The settings apply to the specified type of cache. **C or c** Combined data and instruction cache **D or d** Data cache **I or i** Instruction cache

If you specify the wrong values for the cache configuration or run the program on a machine with a different configuration, the program will work correctly but may be slightly slower.

You must specify **-O4** or **-qipa** with the **-qcache** option.

Use the following guidelines when specifying **-qcache** suboptions:
- Specify information for as many configuration parameters as possible.
- If the target execution system has more than one level of cache, use a separate **-qcache** option to describe each cache level.
- If you are unsure of the exact size of the cache(s) on the target execution machine, specify an estimated cache size on the small side. It is better to leave some cache memory unused than it is to experience cache misses or page faults from specifying a cache size larger than actually present.
- The data cache has a greater effect on program performance than the instruction cache. If you have limited time available to experiment with different cache configurations, determine the optimal configuration specifications for the data cache first.
- If you specify the wrong values for the cache configuration, or run the program on a machine with a different configuration, program performance may degrade but program output will still be as expected.
- The **-O4** and **-O5** optimization options automatically select the cache characteristics of the compiling machine. If you specify the **-qcache** option together with the **-O4** or **-O5** options, the option specified last takes precedence.

Example
To tune performance for a system with a combined instruction and data level-1 cache, where cache is 2-way associative, 8 KB in size and has 64-byte cache lines, enter:

```
xlC -O4 -qcache=type=c:level=1:size=8;line=64;assoc=2 file.C
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

check

Option Type	Default Value	#pragma options	C	C++
-q*option*	nocheck	CHECK	x	x

Syntax

```
-qcheck | -qcheck=suboptions | -qnocheck
CHECK | CHECK=suboptions | NOCHECK
```

Purpose
Generates code that performs certain types of runtime checking. If a violation is encountered, a runtime exception is raised by sending a **SIGTRAP** signal to the process.

Notes
The **-qcheck** option has the following suboptions. If you use more than one *suboption*, separate each one with a colon (:).

`all`	Switches on all the following suboptions. You can use the **all** option along with the **no...** form of one or more of the other options as a filter. For example, using: `xlC myprogram.c -qcheck=all:nonull` provides checking for everything except for addresses contained in pointer variables used to reference storage. If you use **all** with the **no...** form of the options, **all** should be the first suboption.
`NULLptr | NONULLptr`	Performs runtime checking of addresses contained in pointer variables used to reference storage. The address is checked at the point of use; a trap will occur if the value is less than 512.
`bounds | nobounds`	Performs runtime checking of addresses when subscripting within an object of known size. The index is checked to ensure that it will result in an address that lies within the bounds of the object's storage. A trap will occur if the address does not lie within the bounds of the object.
`DIVzero | NODIVzero`	Performs runtime checking of integer division. A trap will occur if an attempt is made to divide by zero.

Using the **-qcheck** option without any suboptions turns all the suboptions on.

Using the **-qcheck** option with suboptions turns the specified suboptions on if they do not have the no prefix, and off if they have the no prefix.

You can specify the **-qcheck** option more than once. The suboption settings are accumulated, but the later suboptions override the earlier ones.

The **#pragma** options directive must be specified before the first statement in the compilation unit.

The **-qcheck** option affects the runtime performance of the application. When checking is enabled, runtime checks are inserted into the application, which may result in slower execution.

Example
For **-qcheck=null:bounds**:

```
void func1(int* p) {
  *p = 42;            /* Traps if p is a null pointer */
}
void func2(int i) {
  int array[10];
  array[i] = 42;     /* Traps if i is outside range 0 - 9 */
}

For -qcheck=divzero:

void func3(int a, int b) {
  a / b;           /* Traps if b=0  */
}
```

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
Equivalent Batch Compile-Link and Incremental Build Options

chars

Option Type	Default Value	#pragma options	C	C++
-q*option*	chars=unsigned	CHARS=*sign_type*	x	x

Syntax

```
-qchars=signed | -qchars=unsigned
CHARS=signed | CHARS=unsigned
```

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

Notes
You can also specify sign type in your source program using either of the following preprocessor directives:

```
#pragma options chars=sign_type
#pragma chars (sign_type)
```

where *sign_type* is either **signed** or **unsigned**.

Regardless of the setting of this option, the type of **char** is still considered to be distinct from the types **unsigned char** and **signed char** for purposes of type-compatibility checking or C++ overloading.

Example
To treat all **char** types as **signed** when compiling myprogram.c, enter:

```
xlC myprogram.c -qchars=signed
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
Equivalent Batch Compile-Link and Incremental Build Options

compact

Option Type	Default Value	#pragma options	C	C++
-q*option*	nocompact	COMPact	x	x

Syntax

```
-qcompact | -qnocompact
COMPACT | NOCOMPACT
```

Purpose
When used with optimization, reduces code size where possible, at the expense of execution speed.

Notes
Code size is reduced by inhibiting optimizations that replicate or expand code inline. Execution time may increase.

Example
To compile myprogram.c to reduce code size, enter:

```
xlC myprogram.c -qcompact
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

cpluscmt

Option Type	Default Value	#pragma options	C	C++
-q*option*	nocpluscmt	CPLUSCMT	x	

Syntax

```
-qcpluscmt | -qnocpluscmt
CPLUSCMT | NOCPLUSCMT
```

Purpose
Use this option if you want C++ comments to be recognized in C source files.

Notes
The **#pragma options** directive must appear before the first statement in the C language source file and applies to the entire file.

C++ comments have the form **//text**. The two slashes (//) in the character sequence must be adjacent with nothing between them. Everything to the right of them until the end of the logical source line, as indicated by a new-line character, is treated as a comment. The // delimiter can be located at any position within a line.

// comments are *not* part of ANSI C. The result of the following valid ANSI C program will be incorrect if **-qcpluscmt** is specified:

```
main() {
  int i = 2;
  printf("%i\n", i //* 2 */
                 + 1);
}
```

The correct answer is 2 (2 divided by 1). When **-qcpluscmt** is specified, the result is 3 (2 plus 1).

The preprocessor handles all comments in the following ways:
- If the "C" on page 103 option is *not* specified, all comments are removed and replaced by a single blank.
- If the "C" on page 103 option *is* specified, comments are output unless they appear on a preprocessor directive or in a macro argument.
- If "E" on page 116 is specified, continuation sequences are recognized in all comments and are output
- If "P" on page 195 is specified, comments are recognized and stripped from the output, forming concatenated output lines.

A comment can span multiple physical source lines if they are joined into one logical source line through use of the backslash (\) character. You can represent the backslash character by a trigraph (??/).

Example of C++ Comments
The following examples show the use of C++ comments:

```
// A comment that spans two \
   physical source lines
// A comment that spans two ??/
   physical source lines
```

Preprocessor Output Example 1
For the following source code fragment:

```
int a;
int b;  // A comment that spans two \
           physical source lines
int c;
        // This is a C++ comment
int d;
```

The output for the "P" on page 195 option is:

```
int a;
int b;
int c;
int d;
```

The ANSI mode output for the "P" on page 195 "C" on page 103options is:

```
int a;
int b;  // A comment that spans two    physical source lines
int c;
        // This is a C++ comment
int d;
```

The output for the "E" on page 116 option is:

```
int a;
int b;
int c;
int d;
```

The ANSI mode output for the "E" on page 116 "C" on page 103 options is:

```
#line 1 "fred.c"
int a;
int b;  // a comment that spans two \
           physical source lines
int c;
        // This is a C++ comment
int d;
```

Extended mode output for the "P" on page 195 "C" on page 103 options or "E" on page 116 "C" on page 103 options is:

```
int a;
int b;  // A comment that spans two \
           physical source lines
int c;
        // This is a C++ comment
int d;
```

Preprocessor Output Example 2 - Directive Line
For the following source code fragment:

```
int a;
#define mm 1   // This is a C++ comment on which spans two \
                  physical source lines
int b;
               // This is a C++ comment
int c;
```

The output for the "P" on page 195 option is:

```
int a;
int b;
int c;
```

The output for the "P" on page 195"C" on page 103 options:

```
int a;
int b;
               // This is a C++ comment
int c;
```

The output for the "E" on page 116 option is:

```
#line 1 "fred.c"
int a;
#line 4
int b;
int c;
```

The output for the "E" on page 116"C" on page 103 options:

```
#line 1 "fred.c"
int a;
#line 4
int b;
               // This is a C++ comment
int c;
```

Preprocessor Output Example 3 - Macro Function Argument
For the following source code fragment:

```
#define mm(aa) aa
    int a;
    int b;  mm(// This is a C++ comment
                  int blah);
    int c;
            // This is a C++ comment
    int d;
```

The output for the “P” on page 195 option:

```
int a;
int b;  int blah;
int c;
int d;
```

The output for the “P” on page 195“C” on page 103 options:

```
int a;
int b;  int blah;
int c;
        // This is a C++ comment
int d;
```

The output for the “E” on page 116 option is:

```
#line 1 "fred.c"
int a;
int b;
int blah;
int c;
int d;
```

The output for the “E” on page 116“C” on page 103 option is:

```
#line 1 "fred.c"
int a;
int b;
int blah;
int c;
        // This is a C++ comment
int d;
```

A comment may contain a sequence of valid multibyte characters.

The character sequence // begins a C++ comment, except within a header name, a character constant, a string literal, or a comment. The character sequence //, or /* and */ are ignored within a C++ comment. Comments do not nest.

Macro replacement is not performed within comments.

Compile Example
To compile myprogram.c. so that C++ comments are recognized as comments, enter:

```
xlC myprogram.c -qcpluscmt
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Preprocessor Options" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

D

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-Dname=definition | -Dname= | -Dname
```

Purpose Defines the identifier *name* as in a **#define** preprocessor directive. *definition* is an optional definition or value assigned to *name*.

Notes The identifier name can also be defined in your source program using the **#define** preprocessor directive.

> **-D***name***=** is equivalent to `#define name`.
>
> **-D***name* is equivalent to `#define name 1`. (This is the default.)

To aid in program portability and standards compliance, the AIX Version 4 Operating System provides several header files that define macro names you can set with the **-D** option. You can find most of these header files either in the **/usr/include** directory or in the **/usr/include/sys** directory. See "Header Files Overview" in the *AIX Version 4 Files Reference*for more information.

The configuration file uses the **-D** option to specify the following predefined macros:

- _POWER
- _AIX
- _AIX32
- _IBMR2
- _ANSI_C_SOURCE

To ensure that the correct macros for your source file are defined, use the **-D** option with the appropriate macro name. If your source file includes the **/usr/include/sys/stat.h** header file, you must compile with the option **-D_POSIX_SOURCE** to pick up the correct definitions for that file.

If your source file includes the **/usr/include/standards.h** header file, **_ANSI_C_SOURCE**, **_XOPEN_SOURCE**, and **_POSIX_SOURCE** are defined if you have not defined any of them.

The "U" on page 232*name* option has a higher precedence than the **-D***name* option.

Example `To specify that all instances of the name COUNT be replaced by 100 in myprogram.c, enter:`

```
xlC myprogram.c -DCOUNT=100
```

This is equivalent to having **#define COUNT 100** at the beginning of the source file.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
VisualAge C++ Predefined Macro Names
"Options that Specify Preprocessor Options" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

datalocal, dataimported

Option Type	Default Value	#pragma options	C	C++
-q*option*	dataimported	DATALOCal, DATAIMPorted	x	x

Syntax

```
-qdatalocal | -qdatalocal=names
-qdataimported | -qdataimported=names

DATALOCAL | DATALOCAL=names
DATAIMPORTED | DATAIMPORTED=names
```

Purpose
Mark data as local or imported.

Notes

Local variables	are statically bound with the functions that use them. **-qdatalocal** changes the default to assume that all variables are local. **-qdatalocal=***names* marks the named variables as local, where *names* is a list of identifiers separated by colons (**:**). The default is not changed. Performance may decrease if an imported variable is assumed to be local.
Imported variables	are dynamically bound with a shared portion of a library. **-qdataimported** changes the default to assume that all variables are imported. **-qdataimported=***names* marks the named variables as imported, where *names* is a list of identifiers separated by colons (**:**). The default is not changed.

Conflicts among the data-marking options are resolved in the following manner:

Options that list variable names	The last explicit specification for a particular variable name is used.
Options that change the default	This form does not specify a name list. The last option specified is the default for variables not explicitly listed in the name-list form.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

dbxextra

Option Type	Default Value	#pragma options	C	C++
-q*option*	nodbxextra	-	x	

Syntax

```
-qdbxextra | -qnodbxextra
```

Purpose
Specifies that all **typedef** declarations, **struct**, **union**, and **enum** type definitions are included for debugging.

Notes
Use this option with the "g" on page 135 option to produce additional debugging information for use with the IBM Distriuted Debugger or **xldb**.

When you specify the "g" on page 135 option, debugging information is included in the object file. To minimize the size of object and executable files, the compiler only includes information for symbols that are referenced. Debugging information is not produced for unreferenced arrays, pointers, or file-scope variables unless **-qdbxextra** is specified.

Using **-qdbxextra** may make your object and executable files larger.

Example
To include all symbols in myprogram.c for debugging, enter:

```
xlC myprogram.c -g -qdbxextra
```

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
Equivalent Batch Compile-Link and Incremental Build Options

digraph

Option Type	Default Value	#pragma options	C	C++
-q*option*	C nodigraph C++ digraph	-	x	x

Syntax

```
-qdigraph | -qnodigraph
```

Purpose
Lets you use digraph key combinations or keywords to represent characters not found on some keyboards. Digraphs are enabled by default.

Notes
A digraph is a keyword or combination of keys that lets you produce a character that is not available on all keyboards.

The digraph key combinations are:

Key Combination	Character Produced
<%	{
%>	}
<:	[
:>	]
%%	#

Additional keywords, valid in C++ programs only, are:

Keyword	Character Produced
`bitand`	&
`and`	&&
`bitor`	|
`or`	||
`xor`	ˆ
`compl`	˜
`and_eq`	&=
`or_eq`	|=
`xor_eq`	ˆ=
`not`	!
`not_eq`	!=

Example
To disable digraph character sequences when compiling your program, enter:

```
xlC myprogram.c -qnodigraph
```

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

dollar

Option Type	Default Value	#pragma options	C	C++
-q*option*	nodollar	-	x	x

Syntax

```
-qdollar | -qnodollar
```

Purpose
Allows the **$** symbol to be used in the names of identifiers.

Example
To compile myprogram.c so that **$** is allowed in identifiers in the program, enter:

```
xlC myprogram.c -qdollar
```

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

E

Option Type	Default Value	#pragma options	C	C++
-type	-	-	x	x

Syntax

```
-E
```

Purpose
Runs the source files named in the compiler invocation through the preprocessor. The **-E** option calls the preprocessor directly as **/usr/vacpp/exe/xlCcpp**.

Notes
The **-E** and "P" on page 195 options have different results. When the **-E** option is specified, the compiler assumes that the input is a C or C++ file and that the output will be recompiled or reprocessed in some way. These assumptions are:

- Original source coordinates are preserved. This is why #line directives are produced.
- All tokens are output in their original spelling, which, in this case, includes continuation sequences. This means that any subsequent compilation or reprocessing with another tool will give the same coordinates (for example, the coordinates of error messages).

The "P" on page 195 option is used for general-purpose preprocessing. No assumptions are made concerning the input or the intended use of the output. This mode is intended for use with input files that are not written in C or C++. As such, all preprocessor-specific constructs are processed as described in the ANSI C standard. In this case, the continuation sequence is removed as described in the "Phases of Translation" of that standard. All non-preprocessor-specific text should be output as it appears.

Using **-E** causes **#line** directives to be generated to preserve the source coordinates of the tokens. Blank lines are stripped and replaced by compensating **#line** directives.

The line continuation sequence is removed and the source lines are concatenated with the "P" on page 195 option. With the **-E** option, the tokens are output on separate lines in order to preserve the source coordinates. The continuation sequence may be removed in this case.

The **-E** option overrides the "P" on page 195, "o" on page 190, and "syntaxonly (C Only)" on page 225 options, and accepts any file name.

If used with the "M" on page 177 option, **-E** will work only for files with a **.C** (C++ source files), **.c** (C source files), or a **.i** (preprocessed source files) filename suffix. Source files with unrecognized filename suffixes are treated and preprocessed as C files, and no error message is generated.

Unless **-C** is specified, comments are replaced in the preprocessed output by a single space character. New lines and **#line** directives are issued for comments that span multiple source lines, and when "C" on page 103 is not specified. Comments within a macro function argument are deleted.

The default is to preprocess, compile, and link-edit source files to produce an executable file.

Example
To compile myprogram.c and send the preprocessed source to standard output, enter:

```
xlC myprogram.c -E
```

If myprogram.c has a code fragment such as:

```
#define SUM(x,y) (x + y) ;
int a ;
#define mm 1 ; /* This is a comment in a
                  preprocessor directive */
int b ;        /* This is another comment across
                  two lines */
int c ;
               /* Another comment */
c = SUM(a, /* Comment in a macro function argument*/
        b) ;
```

the output will be:

```
#line 2 "myprogram.c"
int a;
#line 5
int b;
int c;
c =
(a + b);
```

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

eh (C++ Only)

Option Type	Default Value	#pragma options	C	C++
-q*option*	eh			x

Syntax

```
-qeh | -qnoeh
```

Purpose
Controls whether exception handling is enabled in the module being compiled.

If your program does not use C++ structured exception handling, compile with **-qnoeh** to prevent generation of code that is not needed by your application.

If your program uses C++ exception handling, the program behaviour is undefined if -qnoeh is specified

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

enum

Option Type	Default Value	#pragma options	C	C++
-q*option*	enum=int	ENUM=*suboption*	x	x

Syntax

```
-qenum=small | -qenum=int | -qenum=intlong | -qenum=1 | -qenum=2 | -qenum=4 | -qenum=8
ENUM=SMALL | ENUM=INT | ENUM=INTLONG | ENUM=1 | ENUM=2 | ENUM=4 | ENUM=8 | ENUM=RESET
```

Purpose

Specifies the amount of storage occupied by enumerations.

Notes

Valid *suboptions* are:

`-qenum=small`	Specifies that enumerations occupy a minimum amount of storage: either 1, 2, or 4 bytes of storage, depending on the range of the **enum** constants. In 64-bit compilation mode, the enumerations can also use 8 bytes of storage.
`-qenum=int`	Specifies that enumerations occupy 4 bytes of storage and are represented by **int**.
C++ `-qenum=intlong`	Valid only in 64-bit compiler mode. Specifies that enumerations occupy 8 bytes of storage and are represented by **long**, if **-q64** is specified and the range of the **enum** constants exceed the limit for **int**. Otherwise, the enumerations occupy 4 bytes of storage and are represented by int.
`-qenum=1`	Specifies that enumerations occupy 1 byte of storage.
`-qenum=2`	Specifies that enumerations occupy 2 bytes of storage.
`-qenum=4`	Specifies that enumerations occupy 4 bytes of storage.
`-qenum=8`	Valid only in 64-bit compiler mode. Specifies that enumerations occupy 8 bytes of storage.

RESET	Valid in **#pragma enum** statement only. Resets the **enum** mapping rule to the rule that was in effect before the current mapping rule. If no previous enum mapping rule was specified in the file, the rule specified when the compiler was initially invoked is used.

The **enum** *constants* are always of type **int**, except for the following cases:

- If **-q64** is not specified, and if the range of these constants is beyond the range of **int**, **enum** constants will have type **unsigned int** and be 4 bytes long.
- If **-q64** is specified, and if the range of these constants is beyond the range of **int**, **enum** constants will have type **long** and be 8 bytes long.

The **-qenum=small** option allocates to an **enum** *variable* the amount of storage that is required by the smallest predefined type that can represent that range of **enum** constants. By default, an unsigned predefined type is used. If any **enum** constant is negative, a signed predefined type is used.

The **-qenum=1|2|4|8** options allocate a specific amount of storage to an **enum** *variable*. If the specified storage size is smaller than that required by the range of **enum** variables, the requested size is kept but a warning is issued. For example:

```
enum {frog, toad=257} amph;
1506-387 (W) The enum cannot be packed to the requested size.
        Use a larger value for -qenum.
(The enum size is 1 and the value of toad is 1)
```

For each **#pragma options enum=** directive that you put in a source file, it is good practice to have a corresponding **#pragma options enum=reset** before the end of that file. This is the only way to prevent one file from potentially changing the **enum=** setting of another file that **#include**s it. The #pragma enum() directive can be instead of **#pragma options enum=**. The two pragmas are interchangeable.

The tables below show the priority for selecting a predefined type. They also shows the the predefined type, the maximum range of **enum** constants for the corresponding predefined type, and the amount of storage that is required for that predefined type (that is, the value that the **sizeof** operator would yield when applied to the minimum-sized **enum**).

In 32-bit compilation mode:

	enum=int		enum=small		enum=1		enum=2		enum=4	
Range	variable	constant	variable	constant	variable	constant	variable	constant	variable	constant
0..127	int	int	unsigned char	int	signed char	int	short	int	int	int
-128..127	int	int	signed char	int	signed char	int	short	int	int	int
0..255	int	int	unsigned char	int	unsigned char	int	short	int	int	int
0..32767	int	int	unsigned short	int	unsigned short[1]	int	short	int	int	int
-32768..32767	int	int	short	int	short[1]	int	short	int	int	int

0..65535	int	int	unsigned short	int	unsigned short[1]	int	unsigned short	int	int	int
0..2147483647	int	int	unsigned int	unsigned int	unsigned int[1]	int	unsigned int[1]	int	int	int
-(2147483647+1)..2147483647	int	int	int	int	int[1]	int	int[1]	int	int	int
0..4294967295	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int[1]	unsigned int	unsigned int[1]	unsigned int	unsigned int	unsigned int

Notes:
1. These enumerations are too large to the particular enum=1|2|4 option. The size of the enum is increased to hold the entire range of values. It is recommended that you change the enum option to match the size of the enum required.

In 64-bit compilation mode:

	enum=int		enum=intlong C++		enum=small		enum=1		enum=2		enum=4		enum=8	
Range	var	const	var	const	var	const	var	const	var	const	var	const	var	const
0..127	int	int	int	int	unsigned char	int	signed char	int	short	int	int	int	long	long
-128..127	int	int	int	int	signed char	int	signed char	int	short	int	int	int	long	long
0..255	int	int	int	int	unsigned char	int	unsigned char	int	short	int	int	int	long	long
0..32767	int	int	int	int	unsigned short	int	unsigned short[1]	int	short	int	int	int	long	long
-32768..32767	int	int	int	int	short	int	short[1]	int	short	int	int	int	long	long
0..65535	int	int	int	int	unsigned short	int	unsigned short[1]	int	unsigned short	int	int	int	long	long
0..2147483647	int	int	int	int	unsigned int	unsigned int	int[1]	int	int[1]	int	int	int	long	long
-(2147483647+1)..2147483647	int	int	int	int	int	int	int[1]	int	int[1]	int	int	int	long	long
0..4294967295	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int[1]	unsigned int	unsigned int[1]	unsigned int	unsigned int	unsigned int	long	long
0..(2^{63}-1)	ERR[2]	ERR[2]	long	long	unsigned long	unsigned long	long[1]	long	long[1]	long	long[1]	long	long	long
-2^{63}..(2^{63}-1)	ERR[2]	ERR[2]	long	long	long	long	long[1]	long	long[1]	long	long[1]	long	long	long
0..2^{64}	ERR[2]	ERR[2]	unsigned long	unsigned long	unsigned long	unsigned long	unsigned long[1]	unsigned long	unsigned long[1]	unsigned long	unsigned long[1]	unsigned long	unsigned long	unsigned long

Notes:
1. These enumerations are too large to the particular enum=1|2|4|8 option. The size of the enum is increased to hold the entire range of values. It is recommended that you change the enum option to match the size of the enum required.
2. These enumerations are too large for the enum=int option. It is recommended that you change reduce the range of the values of the enumerations or change the enum option to enum=intlong.

The following are invalid enumerations or invalid usage of **#pragma options enum=**:

- You cannot change the storage allocation of an enum using a **#pragma options 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 options enum=small
enum e_tag {
  a,
  b,
#pragma options enum=int /* error: cannot be within a declaration */
  c
} e_var;
#pragma options enum=reset /* second reset isn't required */
```

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

- The **enum** constant range does not fit within the range of an **unisgned int**.

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

A **-qenum=reset** option corresponding to the **#pragma options enum=reset** directive does not exist. Attempting to use **-qenum=reset** generates a warning message and the option is ignored.

Examples

1. One typical use for the **reset** suboption is to reset 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 1
/*
 * File small_enum.h
 * This enum must fit within an unsigned char type
 */
#pragma options enum=small
   enum e_tag {a, b=255};
   enum e_tag u_char_e_var; /* occupies 1 byte of storage */

/* Reset the enumeration size to whatever it was before */
#pragma options enum=reset
#endif
```

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

```
/*
 * File int_file.c
 * Defines 4 byte enums
 */
#pragma options 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 reset the
 * enum size
 */
enum sushi {CALIF_ROLL, SALMON_ROLL, TUNA, SQUID, UNI};
enum sushi first_order = UNI;
```

The enumerations **test_enum** and **test_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.

2. If the following C fragment 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**.
3. If the following C code fragment 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)** smaller, it will be used to represent the **enum**.
4. If you compile a file myprogram.c using the command:

   ```
   xlC myprogram.c -qenum=small
   ```

   assuming file myprogram.c does not contain **#pragma options=int** statements, all **enum** variables within your source file will occupy the minimum amount of storage.
5. If you compile a file yourfile.c that contains the following lines:

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

   #pragma options enum=small
     enum sushi {CALIF_ROLL, SALMON_ROLL, TUNA, SQUID, UNI};
     enum sushi first_order = UNI;

   #pragma options enum=int
     enum music {ROCK, JAZZ, NEW_WAVE, CLASSICAL};
     enum music listening_type;
   ```

   using the command:

   ```
   xlC yourfile.c
   ```

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

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

maxerr

Option Type	Default Value	#pragma options	C	C++
-q*option*	nomaxerr	-	x	x

Syntax

```
-qmaxerr=num:[sev_level] | -qnomaxerr
```

Purpose
Instructs the compiler to halt compilation when *num* errors of severity *sev_level* or higher is reached.

Notes
num must be an integer. *sev_level* must be one of the following:

sev_level	Description
i	Informational
w	Warning
e	Error
s	Severe error

If no value is specified for *sev_level*, the current value of the "halt" on page 138 option is used. The default value for **-qhalt** is **s** (severe error).

If the **-qmaxerr** option is specified more than once, the **-qmaxerr** option specified last determines the action of the option. If both the **-qmaxerr** and "halt" on page 138 options are specified, the **-qmaxerr** or **-qhalt**option specified last determines the severity level used by the **-qmaxerr** option.

An unrecoverable error occurs when the number of errors reached the limit specified. The error message issued is similar to:

```
1506-672 (U) The number of errors has reached the limit of ...
```

If **-qnomaxerr** is specified, the entire source file is compiled regardless of how many errors are encountered.

Diagnostic messages may be controlled by the "flag" on page 126 option.

Examples
1. To stop compilation of myprogram.c when 10 warnings are encounted, enter the command:
   ```
   xlC myprogram.c -qmaxerr=10:w
   ```
1. To stop compilation of myprogram.c when 5 severe errors are encounted, assuming that the current "halt" on page 138 option value is **S** (severe), enter the command:
   ```
   xlC myprogram.c -qmaxerr=5
   ```
1. To stop compilation of myprogram.c when 3 informationals are encountered, enter the command:
   ```
   xlC myprogram.c -qmaxerr=3:i
   ```

or:

```
xlC myprogram.c -qmaxerr=5:w qmaxerr=3 -qhalt=i
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
"Message Severity Levels and Compiler Response" on page 44
Equivalent Batch Compile-Link and Incremental Build Options

extchk

Option Type	Default Value	#pragma options	C	C++
-q*option*	noextchk	EXTCHK	x	x

Syntax

```
-qextchk | -qnoextchk
EXTCHK | NOEXTCHK
```

Purpose
Generates bind-time type checking information and checks for compile-time consistency.

Notes
-qextchk checks for consistency at compile time and detects mismatches across compilation units at link time.

-qextchk does not perform type checking on functions or objects that contain references to incomplete types.

Example
To compile myprogram.c so that bind-time checking information is produced, enter:

```
xlC myprogram.c -qextchk
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

F

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-Fconfig_file:stanza | -Fconfig_file | -F:stanza
```

Purpose
Names an alternative batch configuration file (.cfg) for **xlC**.

Notes

config_file	Specifies the configuration of your system to the compiler.
stanza	Is the name of the command used to invoke the compiler. This directs the compiler to the config_file under stanza for the description of the compiler environment. This suboption is not required.

The default is a batch configuration file supplied at installation time called /etc/vac.cfg. Any file names or stanzas that you specify on the command line or within your source file override the defaults specified in the **/etc/vac.cfg** batch configuration file.

For information regarding the contents of the batch configuration file, refer to "Specify Batch Compiler Options in a Configuration File" on page 19.

The "B" on page 99, "t" on page 227, and "W" on page 239 options override the **-F** option.

Example
To compile myprogram.c using a batch configuration file **/usr/tmp/myvacpp.cfg** with an **xlC** stanza, enter:

```
xlC myprogram.c -F/usr/tmp/myvac.cfg:xlC
```

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
Equivalent Batch Compile-Link and Incremental Build Options

f

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-ffilelistname
```

Purpose
Names a file to store a list of object files for **xlC** to pass to the linker. The *filelistname* file should contain only the names of object files. There should be one object file per line. This option is the same as the -f option for the **ld** command.

Example
To pass the list of files contained in `myobjlistfile` to the linker, enter:

```
xlC -f/usr/tmp/myobjlistfile
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Linkage Options" on page 42
Equivalent Batch Compile-Link and Incremental Build Options

fdpr

Option Type	Default Value	#pragma options	C	C++
-q*option*	nofdpr	-	x	x

Syntax

```
-qfdpr | -qnofdpr
```

Purpose
Collects information about your program for use with the AIX **fdpr** (Feedback Directed Program Restructuring) performance-tuning utility.

Notes
You should compile your program with **-qfdpr** before optimizing it with the **fdpr** performance-tuning utility. Optmization data is stored in the object file.

For more information on using the **fdpr** performance-tuning utilty, refer to the *AIX Version 4 Commands Reference* or enter the command:

```
man fdpr
```

Example
To compile myprogram.c so it include data required by the **fdpr** utility, enter:

```
xlC myprogram.c -fdpr
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

flag

Option Type	Default Value	#pragma options	C	C++
-q*option*	flag=i:i	FLAG=*severity1:severity2*	x	x

Syntax

```
-qflag=severity1:severity2
FLAG=severity1:severity2
```

Purpose
Specifies the minimum severity level of diagnostic messages to be reported in a listing and displayed on a terminal.

Notes

severity1	Message level reported in listing
severity2	Message level reported on terminal

You must specify a level for both *severity1* and *severity2*.

Diagnostic messages have the following severity levels:

severity	**Description**
i	Information
w	Warning
e	Error
s	Severe error
u	Unrecoverable error

Specifying informational messages does not turn on the "info" on page 144 option.

Example
To compile myprogram.c so that the listing shows all messages that were generated and your workstation displays only error and higher messages, enter:

```
xlC myprogram.c -qflag=I:E
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
"Message Severity Levels and Compiler Response" on page 44
Equivalent Batch Compile-Link and Incremental Build Options

float

Option Type	Default Values	#pragma options	C	C++
-q*option*	noemulate nofltint fold nohsflt nohssngl maf norndsngl norrm norsqrt nospnans	FLOAT	x	x

Syntax:

```
-qfloat=suboptions
FLOAT=suboptions
```

Purpose
Specifies various floating-point options. These options provide different strategies for speeding up or improving the accuracy of floating-point calculations.

Notes
Using the **float** option may produce results that are not precisely the same as the default. Incorrect results may be produced if not all required conditions are met. For these reasons, you should only use this option if you are experienced with floating-point calculations involving IEEE floating-point values and can properly

assess the possibility of introducing errors in your program. See Using Floating-Point Options on the RISC System/6000 System before using this option.

The **float** option has the following *suboptions*. If you use more than one suboption, separate each one with a colon (:).

`-qfloat=emulate | -qfloat=noemulate`	Emulates the floating-point instructions omitted by the PowerPC 403™ processor. The default is **float=noemulate**. To emulate PowerPC 403 processor floating-point instructions, use **-qfloat=emulate**. Function calls are emitted in place of PowerPC 403 floating-point instructions. Use this option only in a single-threaded, stand-alone environment targeting the PowerPC 403 processor. Do not use **-qfloat=emulate** with any of the following: • **“arch” on page 96=pwr**, **-qarch=pwr2**, **-qarch=pwrx** • “ldbl128, longdouble” on page 173, “ldbl128, longdouble” on page 173 • **xlC128** or **xlc128** compiler invocation commands
`-qfloat=fltint | -qfloat=nofltint`	Speeds up floating-point-to-integer conversions by using faster inline code that does not check for overflows. The default is **float=nofltint**, which checks floating-point-to-integer conversions for out-of-range values. This suboption must only be used with an optimization option. • For “O, optimize” on page 192, the default is **-qfloat=nofltint**. • For “O, optimize” on page 192, the default is **-qfloat=fltint**. To include range checking in floating-point-to-integer conversions with the “O, optimize” on page 192 option, specify **-qfloat=nofltint**. • “strict” on page 221 sets **-qfloat=fltint** Changing the optimization level will not change the setting of the **fltint** option if **fltint** has already been specified. If the **“strict” on page 221 | “strict” on page 221** and **-qfloat=** options conflict, the last setting is used.
`-qfloat=fold | -qfloat=nofold`	Specifies that constant floating-point expressions are to be evaluated at compile time rather than at run time. The **-qfloat=fold** option replaces the obsolete “fold” on page 133 option. Use **-qfloat=fold** in your new applications.

`-qfloat=hsflt | -qfloat=nohsflt`	Speeds up calculations by enforcing the rounding of computed values to single precision before storing and on conversions from floating point to integer. **nohsflt** specifies that single-precision expressions are rounded after expression evaluation and that floating-point-to-integer conversions are to be checked for out-of-range values. The **hsflt** option overrides the **rndsngl**, **nans**, and **spnans** options. **Note:** The **hsflt** option is for specific applications in which floating-point computations have known characteristics. Using this option when you are compiling other application programs can produce incorrect results without warning. The **-qfloat=hsflt** option replaces the obsolete "hsflt" on page 140 option. Use **-qfloat=hsflt** in your new applications.
`-qfloat=hssngl | -qfloat=nohssngl`	Specifies that single-precision expressions are rounded only when the results are stored into **float** memory locations. **nohssngl** specifies that single-precision expressions are rounded after expression evaluation. Using **hssngl** can improve runtime performance but is safer than using **-qfloat=hsflt**. The **-qfloat=hssngl** option replaces the obsolete "hssngl" on page 141 option. Use **-qfloat=hssngl** in your new applications.
`-qfloat=maf | -qfloat=nomaf`	Makes floating-point calculations faster and more accurate by using floating-point multiply-add instructions where appropriate. The results may not be exactly equivalent to those from similar calculations performed at compile time or on other types of computers. This option may affect the precision of floating-point intermediate results. The **-qfloat=maf** option replaces the obsolete "maf" on page 181 option. Use **-qfloat=maf** in your new applications.

`-qfloat=nans | -qfloat=nonans`	Generates extra instructions to detect signalling NaN (Not-a-Number) when converting from single precision to double precision at run time. The option **nonans** specifies that this conversion need not be detected. **-qfloat=nans** is required for full compliance to the IEEE 754 standard. The **hsflt** option overrides the **nans** option. When used with the "flttrap" on page 131 or **-qflttrap=invalid** option, the compiler detects invalid operation exceptions in comparison operations that occur when one of the operands is a signalling NaN. The **-qfloat=nans** option replaces the obsolete **-qfloat=spnans** option and the "spnans" on page 218 option. Use **-qfloat=nans** in your new applications.
`qfloat=rndsngl | -qfloat=norndsngl`	Specifies that the result of each single-precision (**float**) operation is to be rounded to single precision. **-qfloat=norndsngl** specifies that rounding to single-precision happens only after full expressions have been evaluated. Using this option may sacrifice speed for consistency with results from similar calculations on other types of computers. The **hsflt** option overrides the **rndsngl** option. The **-qfloat=rndsngl** option replaces the obsolete "rndsngl" on page 209 option. Use **-qfloat=rndsngl** in your new applications.
`-qfloat=rrm | -qfloat=norrm`	Prevents floating-point optimizations that are incompatible with runtime rounding to plus and minus infinity modes. Informs the compiler that the floating-point rounding mode may change at run time or that the floating-point rounding mode is not *round to nearest* at run time. **-qfloat=rrm** must be specified if the Floating Point Status and Control register is changed at run time (as well as for initializing exception trapping). The **-qfloat=rrm** option replaces the obsolete "rrm" on page 211 option. Use **-qfloat=rrm** in your new applications.

`-qfloat=rsqrt | -qfloat=norsqrt`	Specifies whether a sequence of code that involves division by the result of a square root can be replaced by calculating the reciprocal of the square root and multiplying. Allowing this replacement produces code that runs faster. • For "O, optimize" on page 192, the default is **-qfloat=norsqrt**. • For "O, optimize" on page 192, the default is **-qfloat=rsqrt**. Use **-qfloat=norsqrt** to override this default. • "strict" on page 221 sets **-qfloat=rsqrt**. (Note that **-qfloat=rsqrt** means that **errno** will *not* be set for any **sqrt** function calls.) • **-qfloat=rsqrt** has no effect when **-qarch=pwr2** is also specified. • **-qfloat=rsqrt** has no effect unless **-qignerrno** is also specified. Changing the optimization level will not change the setting of the **rsqrt** option if **rsqrt** has already been specified. If the **"strict" on page 221 | "strict" on page 221** and **-qfloat=** options conflict, the last setting is used.
`-qfloat=spnans | -qfloat=nospnans`	Generates extra instructions to detect signalling NaN on conversion from single precision to double precision. The option **nospnans** specifies that this conversion need not be detected. The **hsflt** option overrides the **spnans** option. The **-qfloat=nans** option replaces the obsolete **-qfloat=spnans** and "spnans" on page 218 options. Use **-qfloat=nans** in your new applications.

Example
To compile myprogram.c so that range checking occurs and multiply-add instructions are not generated, enter:

```
xlC myprogram.c -qfloat=fltint:nomaf
```

RELATED CONCEPTS

RISC System/6000 Floating Point Hardware

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

flttrap

Option Type	Default Value	#pragma options	C	C++

-q*option*	noflttrap	FLTTRAP	x	x

Syntax:

```
-qflttrap | -qflttrap=suboptions | -qnoflttrap
FLTTRAP | FLTTRAP=suboptions | NOFLTTRAP
```

Purpose
Generates extra instructions to detect and trap floating-point exceptions.

Notes
This option is recognized during linking. **-qnoflttrap** specifies that these extra instructions need not be generated.

If specified with **#pragma options**, the **-qnoflttrap** option *must* be the first option specified.

The **flttrap** option has the following *suboptions*:

`OVerflow`	Generates code to detect and trap floating-point overflow.
`UNDerflow`	Generates code to detect and trap floating-point underflow.
`ZEROdivide`	Generates code to detect and trap floating-point division by zero.
`INValid`	Generates code to detect and trap floating-point invalid operation exceptions.
`INEXact`	Generates code to detect and trap floating-point inexact exceptions.
`ENable`	Enables the specified exceptions in the prologue of the main program. This suboption is required if you want to turn on exception trapping without modifying the source code.
`IMPrecise`	Generates code for imprecise detection of the specified exceptions. If an exception occurs, it is detected, but the exact location of the exception is not determined.

Specifying the **flttrap** option with no suboptions is equivalent to setting **-qflttrap=ov:und:zero:inv:inex**. The exceptions are not automatically enabled, and all floating-point operations are checked to provide precise exception-location information.

If your program contains signalling NaNs, you should use the **"float" on page 127=nans** along with **-qflttrap** to trap any exceptions.

The compiler exhibits behavior as illustrated in the following examples when the **-qflttrap** option is specified together with "O, optimize" on page 192 options:

- with "O, optimize" on page 192:
  - 1/0 generates a **div0** exception and has a result of infinity
  - 0/0 generates an invalid operation
- with "O, optimize" on page 192:
  - 1/0 generates a **div0** exception and has a result of infinity
  - 0/0 returns zero multiplied by the result of the previous division.

Example
To compile myprogram.c so that floating-point overflow and underflow and divide by zero are detected, enter:

```
xlC myprogram.c -qflttrap=overflow:underflow:zerodivide:enable
```

RELATED CONCEPTS

RISC System/6000 Floating Point Hardware

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

fold

Option Type	Default Value	#pragma options	C	C++
-q*option*	fold	FOLD	x	x

Syntax:

```
-qfold | -qnofold
FOLD | NOFOLD
```

Purpose
Specifies that constant floating-point expressions are to be evaluated at compile time.

Notes
This option is obsolete. Use **"float" on page 127=fold** in your new applications.

RELATED CONCEPTS

RISC System/6000 Floating Point Hardware

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

fullpath

Option Type	Default Value	#pragma options	C	C++
-q*option*	nofullpath	-	x	x

Syntax

```
-qfullpath | -qnofullpath
```

Purpose
Specifies what path information is stored for files when you use "g" on page 135.

Notes
Using **-qfullpath** causes the compiler to preserve the absolute (full) path name of source files specified with the **-g** option.

The relative path name of files is preserved when you use **-qnofullpath**.

-qfullpath is useful if the executable file was moved to another directory. If you specified **-qnofullpath**, the debugger would be unable to find the file unless you provide a search path in the debugger. Using **-qfullpath** would locate the file successfully.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
Equivalent Batch Compile-Link and Incremental Build Options

funcsect

Option Type	Default Value	#pragma options	C	C++
-q*option*	nofuncsect		x	x

Syntax

```
-qfuncsect | -qnofuncsect
```

Purpose
Place instructions for each function in a separate object file, control section or csect. By default, each object file will consist of a single control section combining all functions defined in the corresponding source file.

Notes
Using multiple csects increases the size of the object file, but often reduces the size of the final executable by allowing the linkage editor ro remove functions that are not called or that have been inlined by the optimizer at all places they are called. If the file contains initialized static data or

```
#pragma comment copyright
```

some functions will be one machine word larger.

The #pragma options directive must be specified before the first statement in the conpilation unit.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

G

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Purpose
Tells the linkage editor to create a dynamic library. The compiler will automatically export all global symbols from the shared object unless you specify which symbols to export (by using -bE:, -bexport:, -bexpall or -bnoexpall).

If you use -G to create a shared library, the compiler will:

1. If the user doesn't specify -bE:, -bexport:, -bexpall or -bnoexpall, create an export list containing all global symbols using the CreateExportList script. You can specify another script with the -tE/-B or -qpath=E: options.
2. If CreateExportList was used to create the export list and -qexpfile was specified, the export list is saved.
3. Calls the linker with the appropriate options and object files to build a shared object.

This is a linkage editor (**ld**) option. Refer to *AIX Version 4 Commands Reference* for a description of **ld** command usage and synatx.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

g

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-g
```

Purpose
Generates debugging information used by tools such as the IBM Distributed Debugger.

Notes
Avoid using this option with "O, optimize" on page 192 (optimization) option. The information produced may be incomplete or misleading.

If you specify the **-g** option, the inlining option defaults to "Q" on page 206 (no functions are inlined).

The default with **-g** is not to include information about unreferenced symbols in the debugging information.

To include information about both referenced and unreferenced symbols, use the "dbxextra" on page 114 option with **-g**.

To specify that source files used with **-g** are referred to by either their absolute or their relative path name, use "fullpath" on page 133.

You can also use the "linedebug" on page 174 option to produce abbreviated debugging information in a smaller object size.

Example
To compile myprogram.c to produce an executable program **testing** so you can debug it, enter:

```
xlC myprogram.c -o testing -g
```

To compile myprogram.c to produce an executable program **testing_all** containing additional information about unreferenced symbols, so you can debug it, enter:

```
xlC myprogram.c -o testing_all -g -qdbxextra
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
"linedebug" on page 174 Compiler Option
Equivalent Batch Compile-Link and Incremental Build Options

genpcomp

Option Type	Default Value	#pragma options	C	C++
-q*option*	nogenpcomp	-	x	

Syntax

`-qgenpcomp | -qgenpcomp=`*`directory`*` | -qnogenpcomp`

Purpose
Generates a precompiled version of any header file for which the original source file is used. This may help improve compile time when you use the "usepcomp" on page 236 option.

Notes

`-qgenpcomp`	Generates a precompiled header file called **csetc.pch**, and saves it to the current directory.
`-qgenpcomp=`*`directory`*	Generates a precompiled header file. • If *directory* is the name of an existing directory, the precompiled header file is named **csetc.pch** and saved to that named *directory*. • If a directory with the name *directory* does not exist, the precompiled header file is named *directory*, and is saved to the current directory.
`-qnogenpcomp`	Does not generate precompiled header files.

-qgenpcomp and "usepcomp" on page 236 will be ignored if they are both specified along with the **-a** or **-ae** options. Without the **-qusepcomp** option, **-qgenpcomp** is accepted in all cases.

Example
To compile myprogram.c and generate a precompiled header file for any files that have changed since the last compilation, or for any files that do not have precompiled header files, and then place them in the directory **/headers**, enter:

```
xlC myprogram.c -qgenpcomp=/headers
```

The new precompiled header is called **csetc.pch**.

RELATED CONCEPTS

Precompiled C Headers

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

genproto

Option Type	Default Value	#pragma options	C	C++
-q*option*	nogenproto	-	x	

Syntax

```
-qgenproto | -qgenproto=parmnames | -qnogenproto
```

Purpose
Produces ANSI prototypes from K&R function definitions. This should help to ease the transition from K&R to ANSI.

Notes
Using **-qgenproto** without **PARMnames** will cause prototypes to be generated without parameter names. Parameter names are included in the prototype when **PARMnames** is specified.

Example
`For the following function, foo.c:`

```
foo(a,b,c)
  float a;
  int *b;
```

specifying

```
xlC -c -qgenproto foo.c
```

produces

```
int foo(double, int*, int);
```

The parameter names are dropped. On the other hand, specifying

```
xlC -c -qgenproto=parm foo.c
```

produces

```
int foo(double a, int* b, int c);
```

In this case the parameter names are kept.

Note that **float a** is represented as **double** or **double a** in the prototype, since ANSI states that all narrow-type arguments (such as **char**s, **short**s, and **float**s) are widened before they are passed to K&R functions.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
Equivalent Batch Compile-Link and Incremental Build Options

halt

Option Type	Default Value	#pragma options	C	C++
-q*option*	halt=s	HALT=*severity*	x	x

Syntax

```
-qhalt=severity
HALT=severity
```

Purpose
Instructs the compiler to stop after the compilation phase when it encounters errors of specified *severity* or greater.

Notes
severity is one of:

severity	**Description**
i	Information
w	Warning
e	Error
s	Severe error
u	Unrecoverable error

When the compiler stops as a result of the **-qhalt** option, the compiler return code is nonzero.

When **-qhalt** is specified more than once, the lowest severity level is used.

The **-qhalt** option can be overridden by the "maxerr" on page 123 option.

Diagnostic messages may be controlled by the "flag" on page 126 option.

Example
To compile myprogram.c so that compilation stops if a **warning** or higher level message occurs, enter:

```
xlC myprogram.c -qhalt=w
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
"Message Severity Levels and Compiler Response" on page 44
Equivalent Batch Compile-Link and Incremental Build Options

haltonmsg

Option Type	Default Value	#pragma options	C	C++
-q*option*	-	-		x

Syntax

```
-qhaltonmsg=msg_number
```

Purpose
Instructs the compiler to stop after the compilation phase when it encounters the specified *msg_number*.

When the compiler stops as a result of the **-qhaltonmsg** option, the compiler return code is nonzero.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
"Message Severity Levels and Compiler Response" on page 44
Equivalent Batch Compile-Link and Incremental Build Options

heapdebug

Option Type	Default Value	#pragma options	C	C++
-q*option*	noheapdebug		x	x

Syntax

```
-qheapdebug
```

Purpose
Enables debug versions of memory management functions.

Notes
The **-qheapdebug** options specifies that the debug versions of memory management functions (**_debug_calloc**, **_debug_malloc**, **new**, etc.) be used in place of regular memory management functions. This option defines the **__DEBUG_ALLOC__** macro.

By default, the compiler uses the regular memory management functions (**calloc**,**malloc**, **new**, etc.) and does not preinitialize their local storage.

This option makes the C compiler search both usr/vac/include and usr/include.

Example
To compile myprogram.c with the debug versions of memory management functions, enter:

```
xlC -qheapdebug myprogram.c -o testing
```

RELATED CONCEPTS

Debugging Memory Heaps
Memory Management Functions
Managing Memory with Multiple Memory Heaps

RELATED TASKS

Debug Programs with Heap Memory

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
_debug_calloc - Allocate and Initialize Memory
_debug_free - Free Allocated Memory
_debug_heapmin - Free Unused Memory in the Default Heap
_debug_malloc - Allocate Memory
_debug_memcpy - Copy Bytes
_debug_memmove - Copy Bytes
_debug_memset - Set Bytes to Value
_debug_realloc - Reallocate Memory Block
_debug_strcat - Concatenate Strings
_debug_strcpy - Copy Strings
_debug_strncat - Concatenate Strings
_debug_strncpy - Copy Strings
_debug_strnset - Set Characters in String
_debug_strset - Set Characters in String
_debug_ucalloc - Reserve and Initialize Memory from User Heap
_debug_uheapmin - Free Unused Memory in User Heap
_debug_umalloc - Reserve Memory Block from User Heap
_uheap_check - Validate User Heap Memory
_uheapset - Validate and Set Memory Heap
_uheap_walk - Return Information about Memory Heap
Equivalent Batch Compile-Link and Incremental Build Options

hsflt

Option Type	Default Value	#pragma options	C	C++
-q*option*	nohsflt	HSFLT	x	x

Syntax:

```
-qhsflt | -qnohsflt
HSFLT | NOHSFLT
```

Purpose
Speeds up calculations by removing range checking on single-precision **float** results, and on conversions from floating point to integer. **-qnohsflt** specifies that single-precision expressions are rounded after expression evaluation, and that floating-point-to-integer conversions are to be checked for out of range values.

Notes
This option is obsolete. Use "float" on page 127**=hsflt** in your new applications.

The **hsflt** option overrides the "rndsngl" on page 209 and "spnans" on page 218 options.

The **-qhsflt** option is intended for specific applications in which floating-point computations have known characteristics. Using this option when compiling other application programs can produce incorrect results without warning. See RISC System/6000 Floating Point Hardware before you use the **-qhslft** option.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

hssngl

Option Type	Default Value	#pragma options	C	C++
-q*option*	nohssngl	HSSNGL	x	x

Syntax

```
-qhssngl | -qnohssngl
HSSNGL | NOHSSNGL
```

Purpose
Specifies that single-precision expressions are rounded only when the results are stored into **float** memory locations. **nohssngl** specifies that single-precision expressions are rounded after expression evaluation. Using **hssngl** can improve run-time performance.

Notes
This option is obsolete. Use “float” on page 127**=hssngl** in your new applications.

RELATED CONCEPTS

RISC System/6000 Floating Point Hardware

RELATED REFERENCES

“List of Batch Compiler Options and Their Defaults” on page 24
Equivalent Batch Compile-Link and Incremental Build Options

I

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-Idirectory
```

Purpose
Specifies an additional search path if the file name in the **#include** directive is not specified using its absolute path name.

Notes
The value for *directory* must be a valid path name (for example, **/u/golnaz**, or **/tmp**, or **./subdir**). The compiler appends a slash (/) to the directory and then concatenates it with the file name before doing the search. The path directory is the one that the compiler searches first for **#include** files whose names do not start with a slash (/). If directory is not specified, the default is to search the standard directories.

If the **-I**directory option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first.

The **-I**directory option can be specified more than once on the command line. If you specify more than one **-I** option, directories are searched in the order that they appear on the command line. See Directory Search Sequence for Include Files Using Relative Path Names for more information about searching directories.

If you specify a full (absolute) path name on the **#include** directive, this option has no effect.

Example
To compile myprogram.c and search **/usr/tmp** and then **/oldstuff/history** for included files, enter:

```
xlC myprogram.c -I/usr/tmp -I/oldstuff/history
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"idirfirst" Compiler Option
Equivalent Batch Compile-Link and Incremental Build Options

idirfirst

Option Type	Default Value	#pragma options	C	C++
-q*option*	noidirfirst	IDIRFirst	x	x

Syntax

```
-qidirfirst | -qnoidirfirst
IDIRFirst | NOIDIRFirst
```

Purpose
Specifies the search order for files included with the **#include "***file_name***"** directive.

Notes
Use **-qidirfirst** with the **"I" on page 141**directory option.

The normal search order (for files included with the **#include "***file_name***"** directive) *without* the **idirfirst** option is:

1. Search the directory where the current source file resides.
2. Search the directory or directories specified with the **"I" on page 141**directory option.
3. Search the standard include directories, which are:
   - for C programs, **/usr/include**
   - for C++ programs, **/usr/vacpp/include** and **/usr/include**

With **-qidirfirst**, the directories specified with the **"I" on page 141**directory option are searched before the directory where the current file resides.

-qidirfirst has no effect on the search order for the **#include <file_name>** directive.

-qidirfirst is independent of the **-qnostdinc** option. ("stdinc" on page 220 changes the search order for both **#include "file_name"** and **#include <file_name>**.

The search order of files is described in Directory Search Sequence for Include Files Using Relative Path Names.

The last valid **#pragma option [NO]IDIRFirst** remains in effect until replaced by a subsequent **#pragma option [NO]IDIRFirst**.

Example
To compile myprogram.c and search **/usr/tmp/myinclude** for included files before searching the current directory (where the source file resides), enter:

```
xlC myprogram.c -I/usr/tmp/myinclude -qidirfirst
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

ignerrno

Option Type	Default Value	#pragma options	C	C++
-q*option*	noignerrno	-	x	x

Syntax

```
-qignerrno | -qignerrno
```

Purpose
Allows the compiler to perform optimizations that assume **errno** is not modified by system calls.

Notes
Library routines set **errno** when an exception occurs. This setting and subsequent side effects of **errno** may be ignored by specifying **-qignerrno**.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

ignprag

Option Type	Default Value	#pragma options	C	C++
-q*option*	-	IGNPRAG=*suboption*	x	x

Syntax

```
-qignprag=suboption
IGNPRAG=suboption
```

Purpose
Instructs the compiler to ignore the disjoint and/or isolated_call pragmas.

Notes

Suboptions are:

`disjoint`	Ignores all **#pragma disjoint** directives in the source file.
`isolated`	Ignores all **#pragma isolated_call** directives in the source file.

`all`	Ignores all **#pragma isolated_call** and **#pragma disjoint** directives in the source file.
`ibm`	Ignores all IBM parallel processing directives in the source file, such as **#pragma ibm parallel_loop**, **#pragma ibm schedule**.
`omp`	Ignores all OpenMP parallel processing directives in the source file, such as **#pragma omp parallel**, **#pragma omp critical**.

This option is useful for detecting aliasing pragma errors. Incorrect aliasing gives runtime errors that are hard to diagnose. When a runtime error occurs, but the error disappears when you use **-qignprag** with the "O, optimize" on page 192 option, the information specified in the aliasing pragmas is likely incorrect.

Example
To compile myprogram.c and ignore any **#pragma isolated** directives, enter:

```
xlC myprogram.c -qignprag=isolated
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
#pragma disjoint Preprocessor Directive
#pragma isolated_call Preprocessor Directive
"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
Equivalent Batch Compile-Link and Incremental Build Options

info

Option Type	Default Value	#pragma options	C	C++
-q*option*	noinfo	INFO	x	x

Syntax

```
-qinfo | -qinfo=all | -qinfo=suboption[:suboption ...] | -qnoinfo
INFO |  INFO=ALL | INFO=suboption[:suboption ...] | INFO=RESET | NOINFO
```

Purpose
Produces informational messages.

Notes
Specifying **-qinfo** or **-qinfo=all** turns on all diagnostic messages for all groups except for the ppt (preprocessor trace) group in C++ code.

Specifying **-qnoinfo** turns off all diagnostic messages.

You can use the **#pragma options info=***suboption*[:*suboption* **...**] or **#pragma options noinfo** forms of this compiler option to temporarily enable or disable messages in a particular section of program code, and **#pragma options info=reset** to return to your initial **-qinfo** settings.

Available forms of the **-qinfo** option are:

`all`	Turns on all diagnostic messages for all groups. Note: ▶ C The **-qinfo** and **-qinfo=all** forms of the option have the same effect. ▶ C++ The **-qinfo=all** option does not include the `ppt` group (preprocessor trace).
`noinfo`	Turns off all diagnostic messages for specific portions of your program.
▶ C `private`	Lists shared variables made private to a parallel loop.
▶ C `reduction`	Lists all variables that are recognized as reduction variables inside a parallel loop.

group Turns on or off specific groups of messages, where *group* can be one or more of:

group	**Type of messages returned or suppressed**
`cls|nocls`	Classes
`cmp|nocmp`	Possible redundancies in unsigned comparisons
`cnd|nocnd`	Possible redundancies or problems in conditional expressions
`cns|nocns`	Operations involving constants
`cnv|nocnv`	Conversions
`dcl|nodcl`	Consistency of declarations
`eff|noeff`	Statements with no effect
`enu|noenu`	Consistency of enum variables
`ext|noext`	Unused external definitions
`gen|nogen`	General diagnostic messages
`gnr|nognr`	Generation of temporary variables
`got|nogot`	Use of goto statements
`ini|noini`	Possible problems with initialization
`inl|noinl`	Functions not inlined
`lan|nolan`	Language level effects
`obs|noobs`	Obsolete features
`ord|noord`	Unspecified order of evaluation
`par|nopar`	Unused parameters
`por|nopor`	Nonportable language constructs
`ppc|noppc`	Possible problems with using the preprocessor
`ppt|npppt`	Trace of preprocessor actions
`pro|nopro`	Missing function prototypes
`rea|norea`	Code that cannot be reached
`ret|noret`	Consistency of return statements
`trd|notrd`	Possible truncation or loss of data or precision
`tru|notru`	Variable names truncated by the compiler
`uni|nouni`	Unitialized variables
`use|nouse`	Unused auto and static variables
`vft|novft`	Generation of virtual function tables

Example

To compile myprogram.C to produce informational message about all items except conversions and unreached statements, enter:

```
xlC myprogram.C -qinfo=all -qinfo=nocnv:norea
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
#pragma info Preprocessor Directive
Equivalent Batch Compile-Link and Incremental Build Options

initauto

Option Type	Default Value	#pragma options	C	C++
-q*option*	noinitauto	INITAuto	x	x

Syntax

```
-qinitauto=hex_value | -qnoinitauto
INITAUTO=hex_value | NOINITAUTO
```

Purpose
Initializes automatic storage to the two-digit hexadecimal byte value *hex_value*. The option generates extra code to initialize the automatic (stack-allocated) storage of functions. It reduces the runtime performance of the program and should only be used for debugging.

Notes
There is no default setting for the initial value of **-qinitauto**; you must set an explicit value (for example, **-qinitauto=FA**).

Example
To compile myprogram.c so that automatic stack storage is initialized to hex value FF (decimal 255), enter:

```
xlC myprogram.c -qinitauto=FF
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

inlglue

Option Type	Default Value	#pragma options	C	C++
-q*option*	noinlglue	INLGLUE	x	x

Syntax

```
-qinlglue | -qnoinlglue
INLGLUE | NOINLGLUE
```

Purpose
Generates fast external linkage by inlining the pointer glue code necessary to make a call to an external function or a call through a function pointer.

Notes
Glue code, generated by the linker, is used for passing control between two external functions, or when you call functions through a pointer. It is also used to implement C++ virtual function calls. Therefore the **-qinlglue** option only affects function calls through pointers or calls to an external compilation unit. For calls to

an external function, you should specify that the function is imported by using, for example, the “proclocal, procimported, procunknown” on page 202 option.

The inlining of glue code can cause the size of code to grow. This can be overridden by specifying the “compact” on page 108 option, thereby disabling the **-qinlglue** option.

RELATED REFERENCES

“List of Batch Compiler Options and Their Defaults” on page 24
“Options that Specify Linkage Options” on page 42
Equivalent Batch Compile-Link and Incremental Build Options

inline

Option Type	Default Value	#pragma options	C	C++
-q*option*	*See below.*	-	x	x

Syntax

```
-qinline | -qinline=threshold | -qinline-names | -qinline+names | -qinline=limit | -qnoinline
```

Purpose
Attempts to inline functions instead of generating calls to a function. Inlining is performed if possible, but, depending on which optimizations are performed, some functions might not be inlined.

Notes
The **-qinline** option is functionally equivalent to the “Q” on page 206 option.

Because inlining does not always improve run time, you should test the effects of this option on your code. Do not attempt to inline recursive or mutually recursive functions.

Normally, application performance is optimized if you request optimization (“O, optimize” on page 192 option), and compiler performance is optimized if you do not request optimization.

The VisualAge C++ _inline, _Inline, and __inline C language keywords override all **-qinline** options except **-qnoinline**. The compiler will try to inline functions marked with these keywords regardless of other **-qinline** option settings.

To maximize inlining, specify optimization (“O, optimize” on page 192) and also specify the appropriate **-qinline** options.

In the C language, the following **-qinline** options apply:

`-qinline`	The compiler attempts to inline all appropriate functions with 20 executable source statements or fewer, subject to any other settings of the suboptions to the **-qinline** option. If **-qinline** is specified last, all functions are inlined.

`-qinline=`*`threshold`*	Sets a size limit on the functions to be inlined. The number of executable statements must be less than or equal to *threshold* for the function to be inlined. *threshold* must be a positive integer. The default value is 20. Specifying a threshold value of **0** causes no functions to be inlined except those functions marked with the **__inline**, **_Inline**, or **_inline** keywords.

The *threshold* value applies to logical C statements. Declarations are not counted, as you can see in the example below:

```
increment()
{
  int a, b, i;
   for (i=0; i<10; i++) /* statement 1 */
   {
      a=i;              /* statement 2 */
      b=i;              /* statement 3 */
   }
}
```

`-qinline-`*`names`*	The compiler does not inline functions listed by *names*. Separate each *name* with a colon (**:**). All other appropriate functions are inlined. The option implies **-qinline**.

For example:

```
-qinline-salary:taxes:expenses:benefits
```

causes all functions except those named **salary**, **taxes**, **expenses**, or **benefits** to be inlined if possible.

A warning message is issued for functions that are not defined in the source file.

`-qinline+`*`names`*	Attempts to inline the functions listed by *names* and any other appropriate functions. Each *name* must be separated by a colon (**:**). The option implies **-qinline**.
	
	For example,
	`-qinline+food:clothes:vacation`
	
	causes all functions named **food**, **clothes**, or **vacation** to be inlined if possible, along with any other functions eligible for inlining.
	
	A warning message is issued for functions that are not defined in the source file or that are defined but cannot be inlined.
	
	This suboption overrides any setting of the *threshold* value. You can use a threshold value of zero along with **-qinline+***names* to inline specific functions. For example:
	`-qinline=0`
	
	followed by:
	`-qinline+salary:taxes:benefits`
	
	causes *only* the functions named **salary**, **taxes**, or **benefits** to be inlined, if possible, and no others.

`-qinline=`*`limit`*	Specifies the maximum size (in bytes of generated code) to which a function can grow due to inlining. This limit does not affect the inlining of user specified functions.
`-qnoinline`	Does not inline any functions. If **-qnoinline** is specified last, no functions are inlined.

In the C++ language, the following **-qinline** options apply:

`-qinline`	Compiler inlines all functions that it can.
`-qnoinline`	Compiler does not inline any functions.

Default

The default is to treat inline specifications as a hint to the compiler, and the result depends on other options that you select:

- If you specify the "g" on page 135 option (to generate debug information), no functions are inlined.
- If you optimize your program (-O), the compiler attempts to inline all functions declared as
  inline. Otherwise, the compiler attempts to inline some of the simpler functions declared as inline.

Example

To compile myprogram.c so that no functions are inlined, enter:

```
xlC myprogram.c -O -qnoinline
```

To compile myprogram.c so that the compiler attempts to inline functions of fewer than 12 lines, enter:

```
xlc myprogram.c -O -qinline=12
```

RELATED CONCEPTS

Overview of Optimization

RELATED TASKS

Optimize Your Application

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
_inline, _Inline, and __inline (C only)
Keywords in VisualAge C++
Equivalent Batch Compile-Link and Incremental Build Options

ipa (C Only)

Option Type	Default Values		#pragma options	C	C++
	Compile-time	Link-time			
-q*option*	object	noipa	-	x	

Syntax

For compile-time use:

```
-qipa
-qipa=object|noobject
```

For link-time use:

```
-qipa [-qlibansi|-qnolibansi]
-qipa=suboption {, suboption} [-qlibansi|-qnolibansi]
-qnoipa [-qlibansi|-qnolibansi]
```

-qipa Compile-time Formats	Description
-qipa	Activates interprocedural analysis with the following **-qipa** *suboption* defaults: • **inline=auto** • **level=1** • **missing=unknown** • **noprof** • **partition=medium**

-qipa=object -qipa=noobject	Specifies whether to include standard object code in the object files. Specifying the **noobject** suboption ican substantially reduce overall compile time by not generating object code during the first IPA phase. If the "S" on page 213 compiler option is specified with **noobject**, **noobject** is ignored. If compilation and linking are performed in the same step, and neither the **-S** nor any listing option is specified, **-qipa=noobject** is implied by default. If any object file used in linking with **-qipa** was created with the **-qipa=noobject** option, any file containing an entry point (the main program for an executable program, or an exported function for a library) must be compiled with **-qipa**.

-qipa Link-time Formats	**Description**
-qnoipa	Deactivates interprocedural analysis.
-qipa	Activates interprocedural analysis with the following **-qipa** *suboption* defaults: • **inline=auto** • **level=1** • **missing=unknown** • **noprof** • **partition=medium**
-qlibansi -qnolibansi	The **-qlibansi** option assumes that all functions with the name of an ANSI C defined library function are in fact library functions. This is the default setting. The **-qnolibansi** option does not make this assumption.
suboption	Suboptions can take any of the forms shown below. Separate multiple suboptions with commas. <f border="1" cellpadding="5" cellspacing="0">
Suboption	**Description**
exits=*name*{,*name*}	Specifies names of functions which represent program exits. Program exits are calls which can never return and can never call any procedure which has been compiled with IPA pass 1.
inline[=*suboption*]	Same as specifying the "inline" on page 148 compiler option, with *suboption* being any valid **-qinline** suboption.

inline=auto
 	
inline=noauto
	Enables or disables automatic inlining only. The compiler still accepts user-specified functions as candidates for inlining.

inline=*name*{,*name*}	Specifies a comma-separated list of functions to try to inline, where functions are identified by *name*.
noinline=*name*{,*name*}	Specifies a comma-separated list of functions that must not be inlined, where functions are identified by *name*.
inline=limit=*num*	Changes the size limits that the **-Q** option uses to determine how much inline expansion to do. This established limit is the size below which the calling procedure must remain. *number* is the optimizer's approximation of the number of bytes of code that will be generated. Larger values for this number allow the compiler to inline larger subprograms, more subprogram calls, or both. This argument is implemented only when **inline=auto** is on.
inline=threshold=*size*	Specifies the upper size limit of functions to be inlined, where *size* is a value as defined under **inline=limit**. This argument is implemented only when **inline=auto** is on.
isolated=*name*,{*name*}	Specifies a list of *isolated* functions that are not compiled with IPA. Neither isolated functions nor functions within their call chain can refer to global variables.

level=0
 	
level=1
 	
level=2
	Specifies the optimization level for interprocedural analysis. The default level is 1. Valid levels are as follows:
	Level 0 - Does only minimal interprocedural analysis and optimization.
	Level 1 - Turns on inlining, limited alias analysis, and limited call-site tailoring.
	Level 2 - Performs full interprocedural data flow and alias analysis.

list list=[*name*] [short|long]	Specifies that a listing file be generated during the link phase. The listing file contains information about transformations and analyses performed by IPA, as well as an optional object listing generated by the back end for each partition. This option can also be used to specify the name of the listing file. If listings have been requested (using either the "list" on page 175 or -qipa=list options), and *name* is not specified, the listing file name defaults to **a.lst**. The **long** and **short** suboptions can be used to request more or less information in the listing file. The **short** suboption, which is the default, generates the Object File Map, Source File Map and Global Symbols Map sections of the listing. The **long** suboption causes the generation of all of the sections generated through the short suboption, as well as the Object Resolution Warnings, Object Reference Map, Inliner Report and Partition Map sections.
lowfreq=*name*{,*name*}	Specifies names of functions which are likely to be called infrequently. These will typically be error handling, trace, or initialization functions. The compiler may be able to make other parts of the program run faster by doing less optimization for calls to these functions.
missing=*attribute*	Specifies the interprocedural behavior of procedures that are not compiled with **-qipa** and are not explicitly named in an **unknown**, **safe**, **isolated**, or **pure** suboption. The following attributes may be used to refine this information: safe - Functions which do not indirectly call a visible (not missing) function either through direct call or through a function pointer. isolated - Functions which do not directly reference global variables accessible to visible functions. Functions bound from shared libraries are assumed to be *isolated*. pure - Functions which are *safe* and *isolated* and which do not indirectly alter storage accessible to visible functions. *pure* functions also have no observable internal state. unknown - The default setting. This option greatly restricts the amount of interprocedural optimization for calls to *unknown* functions. Specifies that the missing functions are not known to be *safe*, *isolated*, or *pure*.

partition=small
 	
partition=medium
 	
partition=large
 	
partition=*size*
	Specifies the size of each program partition created by IPA during pass 2.
	
	The size of the partition is directly proportional to the time required to link and the quality of the generated code. When partition sizes are large, the time to complete linkage is longer but the quality of the generated code is generally better. An integer may be used to specify partition *size* for finer control. This integer is in terms of unspecified units and its meaning may change from release to release. Its use should be limited to very short term tuning efforts.

pure=*name*{,*name*}	Specifies a list of *pure* functions that are not compiled with **-qipa**. Any function specified as *pure* must be *isolated* and *safe*, and must not alter the internal state nor have side-effects, defined as potentially altering any data visible to the caller.
safe=*name*{,*name*}	Specifies a list of *safe* functions that are not compiled with **-qipa**. Safe functions can modify global variables, but may not call functions compiled with **-qipa**.
unknown=*name*{,*name*}	Specifies a list of *unknown* functions that are not compiled with **-qipa**. Any function specified as *unknown* can make calls to other parts of the program compiled with **-qipa**, and modify global variables and dummy arguments.

filename	Gives the name of a file which contains suboption information in a special format.
	
	The file format is the following:
	`# ... comment`
	`attribute{, attribute} = name{, name}`
	`missing = attribute}, attribute}`
	`exits = name{, name}`
	`lowfreq = name{, name}`
	`inline [ = auto | = noauto ]`
	`inline = name{, name} [ from name{, name}]`
	`inline-threshold = unsigned_integer`
	`inline-limit = unsigned_integer`
	`list [ = file-name | short | long ]`
	`noinline`
	`noinline = name{, name} [ from name{, name}]`
	`level = 0 | 1 | 2`
	`prof [ = file-name ]`
	`noprof`
	`partition = small | medium | large | unsigned_intege`
	
	where *attribute* is one of:
	• exits
	• lowfreq
	• unknown
	• safe
	• isolated
	• pure

Purpose
Turns on or customizes a class of optimizations known as interprocedural analysis (IPA).

Notes
1. IPA can significantly increase compilation time, even with the **-qipa=noobject** option, so using IPA should be limited to the final performance tuning stage of development.
2. Specify the **-qipa** option on both the compile and link steps of the entire application, or as much of it as possible. You should at least compile the file containing **main**, or at least one of the entry points if compiling a library.
3. While IPA's interprocedural optimizations can significantly improve performance of a program, they can also cause previously incorrect but functioning programs to fail. Listed below are some programming practices that can work by accident without aggressive optimization, but are exposed with IPA:
   a. Relying on the allocation order or location of automatics. For example, taking the address of an automatic variable and then later comparing it with the address of another local to determine the growth direction of a stack. The C language does not guarantee where an automatic variable is allocated, or it's position relative to other automatics. Do not compile such a function with IPA(and expect it to work).
   b. Accessing an either invalid pointer or beyond an array's bounds. IPA can reorganize global data structures. A wayward pointer which may have previously modified unused memory may now trample upon user allocated storage.

4. Ensure you have sufficient resources to compile with IPA. IPA can generate significantly larger object files than traditional compilers. As a result, the temporary storage used to hold these intermediate files (by convention /tmp on AIX) is sometimes too small. If a large application is being compiled, consider redirecting temporary storage with the TMPDIR environment variable.
5. Ensure there is enough swap space to run IPA (at least 200Mb for large programs). Otherwise the operating system might kill IPA with a signal 9 , which cannot be trapped, and IPA will be unable to clean up its temporary files.
6. You can link objects created with different releases of the compiler, but you must ensure that you use a linker that is at least at the same release level as the newer of the compilers used to create the objects being linked.
7. Some symbols which are clearly referenced or set in the source code may be optimized away by IPA, and may be lost to debug, nm, or dump outputs. Using IPA together with the **-g** compiler will usually result in non-steppable output.

The necessary steps to use IPA are:

1. Do preliminary performance analysis and tuning before compiling with the **-qipa** option, because the IPA analysis uses a two-pass mechanism that increases compile and link time. You can reduce some compile and link overhead by using the **-qipa=noobject** option.
2. Specify the **-qipa** option on both the compile and the link steps of the entire application, or as much of it as possible. Use suboptions to indicate assumptions to be made about parts of the program *not* compiled with **-qipa**. During compilation, the compiler stores interprocedural analysis information in the **.o** file. During linking, the **-qipa** option causes a complete recompilation of the entire application.

Note: If a Severe error occurs during compilation, **-qipa** returns RC=1 and terminates. Performance analysis also terminates.

Example
To compile a set of files with interprocedural analysis, enter:

```
xlC -c -O3 *.c -qipa
xlC -o product *.o -qipa
```

Here is how you might compile the same set of files, improving the optimization of the second compilation, and the speed of the first compile step. Assume that there exits two functions, *trace_error* and *debug_dump*, which are rarely executed.

```
xlC -c -O3 *.c -qipa=noobject
xlC -c - *.o -qipa=lowfreq=trace_error,debug_dump
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

isolated_call

Option Type	Default Value	#pragma options	C	C++
-q*option*	-	ISOLATED_CALL	x	x

Syntax

```
-qisolated_call=function_name
ISOLATED_CALL=function_name
```

Purpose
Specifies functions in the source file that have no side effects.

Notes

function_name	Is the name of a function that does not have side effects, except changing the value of a variable pointed to by a pointer or reference parameter, or does not rely on functions or processes that have side effects. *Side effects* are any changes in the state of the runtime environment. Examples of such changes are accessing a volatile object, modifying an external object, modifying a file, or calling another function that does any of these things. Functions with no side effects cause no changes to external and static variables. *function_name* can be a list of functions separated by colons (:).

Marking a function as isolated can improve the runtime performance of optimized code by indicating the following to the optimizer:

- external and static variables are not changed by the called function
- calls to the function with loop-invariant parameters may be moved out of loops
- multiple calls to the function with the same parameter may be merged into one call
- calls to the function may be discarded if the result value is not needed

The #pragma options keyword **isolated_call** must be specified at the top of the file, before the first C or C++ statement. You can use the #pragma isolated_call directive at any point in your source file.

Example
To compile myprogram.c, specifying that the functions **myfunction(int)** and **classfunction(double)** do not have side effects, enter:

```
xlC myprogram.c -qisolated_call=myfunction:classfunction
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

keyword

Option Type	Default Value	#pragma options	C	C++
-q*option*	Recognize all C++ keywords	-		x

Syntax

```
-qkeyword=name
-qnokeyword=name
```

Purpose
This option controls whether the specified name is treated as a keyword or an identifier whenever it appears in your C++ source.
By default all the built-in keywords defined in the C++ standard are reserved as keywords.
You cannot add keywords to the C++ language with this option. However, you can use it to enable built-in keywords that have been disabled using -qnokeyword=*name*.
Notes
For example, you can reinstate bool with the following option:

```
xlC -qkeyword=bool
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

L

Option Type	Default Value	#pragma options	C	C++
-flag	*See below.*	-	x	x

Syntax

```
-Ldirectory
```

Purpose
Searches the path directory for library files specified by the "l" on page 160*key* option.

Notes
If the **-L***director*y option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first.

Default
The default is to search only the standard directories.

Example
To compile myprogram.c so that the directory **/usr/tmp/old** and all other directories specified by the -l option are searched for the library **libspfiles.a**, enter:

```
xlC myprogram.c -lspfiles -L/usr/tmp/old
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Linkage Options" on page 42
Equivalent Batch Compile-Link and Incremental Build Options

l

Option Type	Default Value	#pragma options	C	C++
-flag	*See below.*	-	x	x

Syntax

```
-lkey
```

Purpose
Searches the specified library file, lib*key*.so, and then lib*key*.a for dynamic linking, or just *libkey*.a for static linking.

Notes
The actual search path can be modified with the "L" on page 159*directory* or "Z" on page 243 options. See "B" on page 99, "datalocal, dataimported" on page 113, and "bstatic, bdynamic, bshared" on page 102for information on specifying the types of libraries that are searched (for static or dynamic linking).

Default
The default is to search only some of the compiler run-time libraries. See the default batch configuration file for the list of default libraries corresponding to the invocation command being used and the level of the operating system.

Example
To compile myprogram.c and include the Task Library, **libioc.a**, enter:

```
xlC myprogram.c -ltask -lioc
```

RELATED REFERENCES
Default Batch Configuration File
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Linkage Options" on page 42
Equivalent Batch Compile-Link and Incremental Build Options

langlvl

Option Type	Default Value	#pragma options	C	C++
-q*option*	langlvl=ansi*	LANGlvl=*language*	x	x

Syntax

```
-qlanglvl=language
LANGLVL=language
```

Purpose
Selects the C or C++ language level for the compilation.

Default
The default language level is **ansi** when you invoke the compiler using the **xlC**, **xlc**, or **c89** command. The default language level is **extended** when you invoke the compiler using the **cc** command.

You can use either of the following preprocessor directives to specify the language level in your C or C++ source program:

```
#pragma options langlvl=language
#pragma langlvl(language)
```

The **pragma** directive must appear before any noncommentary lines in the source code.

Notes
For C programs, *language* is one of:

`ansi`	Compilation conforms to the ANSI C standard.
`saal2`	Compilation conforms to the SAA C Level 2 CPI language definition, with some exceptions.
`saa`	Compilation conforms to the current SAA C CPI language definition. This is currently SAA C Level 2.
`EXTended`	Provides compatibility with the RT compiler and **classic**.
`classic`	Allows the compilation of non-ANSI programs, and conforms closely to the K&R level preprocessor.
`[no]ucs`	This option controls whether Unicode characters are allowed in identifiers, string literals and character literals in C++ sources. The Unicode character set is supported by the C++ standard. This character set contains the full set of letters, digits and other characters used by a wide range of languages, including all North American and Western European languages. Unicode characters can be 16 or 32 bits. The ASCII one-byte characters are a subset of the Unicode character set. When this option is set to yes, you can insert Unicode characters in your source files either directly or using a notation that is similar to escape sequences. Because many Unicode characters cannot be displayed on the screen or entered from the keyboard, the latter approach is usually preferred. Notation forms for Unicode characters are \u*hhhh* for 16-bit characters, or \U*hhhhhhhh* for 32-bit characters, where *h* represents a hexadecimal digit. Short identifiers of characters are specified by ISO/IEC 10646.

For C++ programs, *language* is one of:

`ansi`	Compilation conforms to the ANSI C standard for C programs, and the proposed ANSI working paper for C++ programs. The macro **__ANSI__** is predefined to be **1**. (Same as `strict98`)
`compat366`	Compilation conforms to the IBM C and C++ Compilers V 3.6.
`extended`	Compilation conforms is the same as **ansi** mode, with some differences.
`strict98`	Compilation conforms to the ANSI C standard for C programs, and the proposed ANSI working paper for C++ programs. The macro **__ANSI__** is predefined to be **1**. (`ansi`)
`[no]anonstruct`	This option controls whether anonymous structs and anonymous classes are allowed in your C++ source.

By default, VisualAge C++ allows anonymous structs. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by Microsoft® Visual C++.

Anonymous structs typically are used in unions, as in the following code fragment:

```
union U {
   struct {
      int i:16;
      int j:16;
   };
   int k;
} u;
// ...
u.j=3;
```

When this option is set, you receive a warning if your code declares an anonymous struct. You can suppress the warning with "suppress" on page 223. When you build with lang(anonymousStructs, no) an anonymous struct is flagged as an error.

Set anonymousStructs to no for compliance with standard C++.

`[no]ansifor`	This option controls whether scope rules defined in the C++ standard apply to names declared in for-init statements. By default, standard C++ rules are used. For example the following code causes a name lookup error: `{ //... for (int i=1; i<5; i++) { cout << i * 2 << endl; } i = 10;  // error }` The reason for the error is that i, or any name declared within a for-init-statement, is visible only within the for statement. To correct the error, either declare i outside the loop or set ansiForStatementScopes to no. Set ansiForStatementScopes to no to allow old language behavior. You may need to do this for code that was developed with other products, such as the compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

`[no]oldfriend`	This option controls whether friend declarations that name classes without elaborated class names are treated as C++ errors.
	
	By default, VisualAge C++ lets you declare a friend class without elaborating the name of the class with the keyword class. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.
	
	For example, the statement below declares the class IFont to be a friend class and is valid when compatFriendDeclarations is set to yes.
	`friend IFont;`
	
	Set compatFriendDeclarations to no for compliance with standard C++. The example declaration above causes a warning unless you modify it to the statement below, or suppress the message with “suppress” on page 223 option.
	`friend class IFont;`

`[no]oldmath`	This option controls which versions of math function declarations in <math.h> are included when you specify math.h as an included or primary source file. By default, the new standard math functions are used. Build with lang(compatMath, no) for strict compliance with the C++ standard. For compatibility with modules that were built with earlier versions of VisualAge C++ and predecessor products you may need to build with lang(compatMath, yes).

The code example under `[no]ansifor`, with its printed layout:

```
{
   //...
   for (int i=1; i<5; i++) {
      cout << i * 2 << endl;
   }
   i = 10;  // error
}
```

`[no]oldtempacc`

This option controls whether access to a copy constructor to create a temporary object is always checked, even if creation of the temporary object is avoided.

By default, VisualAge C++ suppresses the access checking. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, Version 3.5, and Microsoft Visual C++.

When this option is set to yes, you receive a warning if your code uses the extension, unless you disable the message. Disable the message by building with “suppress” on page 223 when the copy constructor is a private member, and -qsuppress=CPPC0307 when the copy constructor is a protected member.

Set -qlanglvl=nooldtempacc for compliance with standard C++. For example, the throw statement in the following code causes an error because the copy constructor is a protected member of class C:

```
class C {
public:
   C(char *);
protected:
   C(const C&);
};
C foo() {return C("test");}  // returns a copy of a C object

void f()
{
// catch and throw both make implicit copies of the thrown object
   throw C("error");         // throws a copy of a C object
   const C& r = foo();       // uses the copy of a C object createc
}
```

The example code above contains three ill formed uses of the copy constructor C(const C&).

`[no]oldtmplalign`

This option specifies the alignment rules implemented in versions of the batch compiler (**xlC**) prior to Version 5.0. These earlier versions of the xlC compiler ignore alignment rules specified for nested templates. By default, these alignment rules are not ignored in VisualAge C++ 4.0 or later. For example, given the following template the size of `A<char>::B` will be 5 with -qlanglvl=nooldtmplalign and 8 with -qlanglvl=oldtmplalign :

```
template <class T>
struct A {
#pragma options align=packed
 struct B {
  T m;
  int m2;
 };
#pragma options align=reset
};
```

[no]oldtmplspec	This option controls whether template specializations that do not conform to the C++ standard are allowed.

By default, VisualAge C++ allows these old specializations (-qlanglvl=nooldtmplspec). This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, Version 3.5, and Microsoft Visual C++.

When -qlanglvl=oldtmplspec is set, you receive a warning if your code uses the extension, unless you suppress the message with "suppress" on page 223.

For example, you can explicitly specialize the template class ribbon for type char with the following lines:

```
template<class T> class ribbon { /*...*/};
class ribbon<char> { /*...*/};
```

Set -qlanglvl=nooldtmplspec for compliance with standard C++. In the example above, the template specialization must be modified to:

```
template<class T> class ribbon { /*...*/};
template<> class ribbon<char> { /*...*/};
```

[no]anonunion	This option controls what members are allowed in anonymous unions.

When this option is set to yes, anonymous unions can have members of all types that standard C++ allows in non-anonymous unions. For example, non-data members, such as structs, typedefs, and enumerations are allowed.

Member functions, virtual functions, or objects of classes that have non-trivial default constructors, copy constructors, or destructors cannot be members of a union, regardless of the setting of this option.

By default, VisualAge C++ allows non-data members in anonymous unions. This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by previous versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

When this option is set to yes, you receive a warning if your code uses the extension, unless you suppress the message with "suppress" on page 223.

Set extendedAnonymousUnions to no for compliance with standard C++.

`[no]illptom`

This option controls what expressions can be used to form pointers to members. VisualAge C++ can accept some forms that are in common use, but do not conform to the C++ standard.

By default, VisualAge C++ allows these forms. This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

When this option is set to yes, you receive warnings if your code uses the extension, unless you suppress the messages with “suppress” on page 223 and “suppress” on page 223.

For example, the following code defines a pointer to a function member, p, and initializes it to the address of C::foo, in the old style:

```
struct C {
void foo(int);
};
void (C::*p) (int) = C::foo;
```

Set illformedPointerToMember to no for compliance with the C++ standard. The example code above must be modified to use the & operator.

```
struct C {
void foo(int);
};
void (C::*p) (int) = &C::foo;
```

`[no]implicitinit`	This option controls whether VisualAge C++ will accept missing or partially specified types as implicitly specifying int. This is no longer accepted in the standard but may exist in legacy code.

With the option set to no, all types must be fully specified.

With the option set to yes, a function declaration at namespace scope or in a member list will implicitly be declared to return int. Also, any declaration specifier sequence that does not completely specify a type will implicitly specify an integer type. Note that the effect is as if the int specifier were present. This means that the specifier const, by itself, would specify a constant integer.

The following specifiers do not completely specify a type.

- auto
- const
- extern
- extern "<literal>"
- inline
- mutable
- friend
- register
- static
- typedef
- virtual
- volatile
- platform specific types (for example, _cdecl, __declspec)

Note that any situation where a type is specified is affected by this option. This includes, for example, template and parameter types, exception specifications, types in expressions (eg, casts, dynamic_cast, new), and types for conversion functions.

By default, VisualAge C++ sets -qlanglvl=implicitinit. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

For example, the return type of function MyFunction is int because it was omitted in the following code:

```
MyFunction()
{
   return 0;
}
```

Set -qlanglvl=noimplicitint for compliance with standard C++. For example, the function declaration above must be modified to:

```
int MyFunction()
{
   return 0;
}
```

`[no]newexcp`

This option determines whether or not the C++ operator new throws an exception. The standard exception std::bad_alloc can be thrown when the requested memory allocation fails. This option does not apply to the nothrow versions of the new operator.

The standard implementation of the new operators fully support exceptions. For compatibility with previous versions of VisualAge C++, these operators return 0 by default.

`[no]offsetnonpod`

This option controls whether the offsetof macro can be applied to classes that are not data-only. C++ programmers often casually call data-only classes “Plain Old Data” (POD) classes.

By default, VisualAge C++ allows offsetof to be used with nonPOD classes. This is an extension to the C++ standard, and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, Version 3.5, and Microsoft Visual C++

When this option is set, you receive a warning if your code uses the extension, unless you suppress the message with “suppress” on page 223.

Set -qlanglvl=nooffsetnonpod for compliance with standard C++.

Set -qlanglvl=offsetnonpod if your code applies offsetof to a class that contains one of the following:

- user-declared constructors or destructors
- user-declared assignment operators
- private or protected non-static data members
- base classes
- virtual functions
- non-static data members of type pointer to member
- a struct or union that has non-data members
- references

`[no]olddigraph`

This option controls whether old-style digraphs are allowed in your C++ source. It applies only when “digraph” on page 114 is also sete.

By default, VisualAge C++ supports only the digraphs specified in the C++ standard.

Set -qlanglvl=olddigraph if your code contains at least one of following digraphs:

Digraph	**Resulting Character**
%%	# (pound sign)
%%%%	## (double pound sign, used as the preprocessor macro concatenation operator)

Set -qlanglvl=noolddigraph for compatibility with standard C++ and the extended C++ language level supported by previous versions of VisualAge C++ and predecessor products.

`[no]trailenum`	This option controls whether trailing commas are allowed in enum declarations. By default, VisualAge C++ allows one or more trailing commas at the end of the enumerator list. This is an extension to the C++ standard, and provides compatibility with Microsoft Visual C++. The following enum declaration uses this extension: `enum grain { wheat, barley, rye,, };` Set -qlanglvl=notrailenum for compliance with standard C++ or with the ANSI language level supported by previous versions of VisualAge C++ and predecessor products.
`[no]typedefclass`	This option provides backwards compatibility with previous versions of VisualAge C++ and predecessor products. The current C++ standard does not allow a typedef name to be specified where a class name is expected. This option relaxes that restriction. Set -qlangllvl=typedefclass to allow the use of typedef names in base specifiers and constructor initializer lists. By default, a typedef name cannot be specified where a class name is expected.
`[no]ucs`	This option controls whether Unicode characters are allowed in identifiers, string literals and character literals in C++ sources. The Unicode character set is supported by the C++ standard. This character set contains the full set of letters, digits and other characters used by a wide range of languages, including all North American and Western European languages. Unicode characters can be 16 or 32 bits. The ASCII one-byte characters are a subset of the Unicode character set. When -qlanglvl=ucs is set, you can insert Unicode characters in your source files either directly or using a notation that is similar to escape sequences. Because many Unicode characters cannot be displayed on the screen or entered from the keyboard, the latter approach is usually preferred. Notation forms for Unicode characters are \u*hhhh* for 16-bit characters, or \U*hhhhhhhh* for 32-bit characters, where *h* represents a hexadecimal digit. Short identifiers of characters are specified by ISO/IEC 10646. The default is -qlanglvl=noucs.

`[no]zeroextarray`	This option controls whether zero-extent arrays are allowed as the last non-static data member in a class definition.

By default, VisualAge C++ allows arrays with zero elements. This is an extension to the C++ standard, and provides compatibility with Microsoft Visual C++. The example declarations below define dimensionless arrays a and b.

```
struct S1 { char a[0]; };
struct S2 { char b[]; };
```

Set zeroExtentArrays to no for compliance with standard C++ or with the ANSI language level supported by previous versions of VisualAge C++ and predecessor products.

When this option is set, you receive warnings about zero-extent arrays in your code, unless you suppress the message with "suppress" on page 223.

Exceptions to the **ansi** mode addressed by **classic** are as follows:

Tokenization

Tokens introduced by macro expansion may be combined with adjacent tokens in some cases. Historically, this was an artifact of the text-based implementations of older preprocessors, and because, in older implementations, the preprocessor was a separate program whose output was passed on to the compiler.

For similar reasons, tokens separated only by a comment may also be combined to form a single token. Here is a summary of how tokenization of a program compiled in **classic** mode is performed:

1. At a given point in the source file, the next token is the longest sequence of characters that can possibly form a token. For example, `i+++++j` is tokenized as `i ++ ++ + j` even though `i ++ + ++ j` may have resulted in a correct program.
2. If the token formed is an identifier and a macro name, the macro is replaced by the text of the tokens specified on its **#define** directive. Each parameter is replaced by the text of the corresponding argument. Comments are removed from both the arguments and the macro text.
3. Scanning is resumed at the first step from the point at which the macro was replaced, as if it were part of the original program.
4. When the entire program has been preprocessed, the result is scanned again by the compiler as in the first step. The second and third steps do not apply here since there will be no macros to replace. Constructs generated by the first three steps that resemble preprocessing directives are not processed as such.

It is in the third and fourth steps that the text of adjacent but previously separate tokens may be combined to form new tokens.

The \ character for line continuation is accepted only in string and character literals and on preprocessing directives.

Constructs such as:

```
#if 0
   "unterminated
#endif
#define US "Unterminating string
char *s = US terminated now"
```

will not generate diagnostic messages, since the first is an unterminated literal in a FALSE block, and the second is completed after macro expansion. However:

```
char *s = US;
```

will generate a diagnostic message since the string literal in `US` is not completed before the end of the line.

Empty character literals are allowed. The value of the literal is sero.

Preprocessing directives

The # token must appear in the first column of the line. The token immediately following # is available for macro expansion. The line can be continued with \ only if the name of the directive and, in the following example, the (has been seen:

```
#define f(a,b) a+b
f\
(1,2)      /* accepted */

#define f(a,b) a+b
f(\
1,2)       /* not accepted */
```

The rules concerning \ apply whether or not the directive is valid. For example,

```
#\
define M 1   /* not allowed */

#def\
ine M 1      /* not allowed */

#define\
M 1          /* allowed */

#dfine\
M 1          /* equivalent to #dfine M 1, even
                though #dfine is not valid  */
```

Following are the preprocessor directive differences between **classic** mode and **ansi** mode. Directives not listed here behave similarly in both modes.

`#ifdef/` `#ifndef`	When the first token is not an identifier, no diagnostic message is generated, and the condition is FALSE.
`#else`	When there are extra tokens, no diagnostic message is generated.
`#endif`	When there are extra tokens, no diagnostic message is generated
`#include`	The < and > are separate tokens. The header is formed by combining the spelling of the < and > with the tokens between them. Therefore /* and // are recognized as comments (and are always stripped), and the ″ and ′ do begin literals within the < and >. (Remember that in C programs, C++-style comments // are recognized when "cpluscmt" on page 108 is specified.)
`#line`	The spelling of all tokens which are not part of the line number form the new file name. These tokens need not be string literals.
`#error`	Not recognized in **classic** mode.
`#define`	A valid macro parameter list consists of zero or more identifiers each separated by commas. The commas are ignored and the parameter list is constructed as if they were not specified. The

Macro expansion	• When the number of arguments on a macro invocation does not match the number of parameters, a warning is issued. • If the (token is present after the macro name of a function-like macro, it is treated as too few arguments (as above) and a warning is issued. • Parameters are replaced in string literals and character literals. • Examples:

```
#define M()    1
#define N(a)   (a)
#define O(a,b) ((a) + (b))

M();  /* no error */
N();  /* empty argument */
O();  /* empty first argument
         and too few arguments */
```

Text Output	No text is generated to replace comments.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
Equivalent Batch Compile-Link and Incremental Build Options

ldbl128, longdouble

Option Type	Default Value	#pragma options	C	C++
-q*option*	noldbl128	LDBL128	x	x

Syntax

```
-qldbl128 | -qnoldbl128 | -qlongdouble | -qnolongdouble
LDBL128 | NOLDBL128 | LONGDOUBLE | NOLONGDOUBLE
```

Purpose
Increases the size of **long double** type from 64 bits to 128 bits.

Notes
The -qlongdouble option is the same as the -qldbl128 option.

Separate libraries are provided that support 128-bit **long double** types. These libraries will be automatically linked if you use any of the invocation commands with the **128** suffix (**xlC128**, **xlc128**, **cc128, xlC128_r**, **xlc128_r,** or **cc128_r**). You can also manually link to the 128-bit versions of the libraries using the "l" on page 160*key* option, as shown in the following table:

Default (64-bit) long double		**128-bit long double**	
Library	**Form of the -l***key* **option**	**Library**	**Form of the -l***key* **option**
libC.a	-lC	libC128.a	-lC128
libC_r.a	-lC_r	libC128_r.a	-lC128_r

Linking without the 128-bit versions of the libraries when your program uses 128-bit **long double**s (for example, if you specify **-qldbl128** alone) may produce unpredictable results.

The **-qldbl128** option defines **__LONGDOUBLE128**.

The **#pragma options** directive must appear before the first C or C++ statement in the source file, and the option applies to the entire file.

Example
To compile myprogram.c so that **long double** types are 128 bits, enter:

```
    xlC myprogram.c -qldbl128 -lC128
```

or:

```
    xlC128 myprogram.c
```

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

libansi

Option Type	Default Value	#pragma options	C	C++
-q*option*	nolibansi	-	x	x

Syntax

```
-qlibansi | -qnolibansi
```

Purpose
Assumes that all functions with the name of an ANSI C library function are in fact the system functions.

Notes
This will allow the optimizer to generate better code because it will know about the behavior of a given function, such as whether or not it has any side effects.

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

linedebug

Option Type	Default Value	#pragma options	C	C++
-q*option*	nolinedebug	-	x	x

Syntax

```
-qLINEDebug | -qNOLINEDebug
```

Purpose
Generates line number and source file name information for the debugger.

Notes
This option produces minimal debugging information, so the resulting object size is smaller than that produced if the "g" on page 135 debugging option is specified. You can use the debugger to step through the source code, but you will not be able to see or query variable information. The traceback table, if generated, will include line numbers.

Avoid using this option with "O, optimize" on page 192 (optimization) option. The information produced may be incomplete or misleading.

If you specify the **-qlinedebug** option, the inlining option defaults to "Q" on page 206 (no functions are inlined).

The "g" on page 135 option overrides the **-qlinedebug** option. If you specify **-g -qnolinedebug** on the command line, **-qnolinedebug** is ignored and the following warning is issued:

```
1506-... (W) Option -qnolinedebug is incompatible with option -g and is ignored.
```

Example
To compile myprogram.c to produce an executable program **testing** so you can step through it with **xldb**, enter:

```
    xlC myprogram.c -o testing -qlinedebug
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
"g" on page 135 Compiler Option
Equivalent Batch Compile-Link and Incremental Build Options

list

Option Type	Default Value	#pragma options	C	C++
-q*option*	nolist	LIST	x	x

Syntax

```
-qlist | -qnolist
LIST | NOLIST
```

Purpose
Produces a compiler listing that includes an object listing.

Notes
For C, options that are not defaults appear in all listings, even if **nolist** is specified. The **noprint** option overrides this option. This does not appply to C++.

Example
To compile myprogram.c to produce an object listing enter:

```
    xlC myprogram.c -qlist
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

longlit

Option Type	Default Value	#pragma options	C	C++
-q*option*	nolonglit		x	x

Syntax

```
-qlonglit | -qnolonglit
```

Purpose
Makes unsuffixed literals into the long type in 64-bit mode. The following table shows the implicit types for constants in 64-bit mode:

	default 64-bit mode	**64-bit mode with qlonglit**
unsuffixed decimal	signed int signed long unsigned long	signed long unsigned long
unsuffixed octal or hex	signed int unsigned int signed long unsigned long	signed long unsigned long
suffixed by u/U	unsigned int unsigned long	unsigned long
suffixed by l/L	signed long unsigned long	signed long unsigned long
suffixed by ul/UL	unsigned long	unsigned long

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

longlong

Option Type	Default Value	#pragma options	C	C++
-q*option*	longlong*	-	x	x

Syntax

```
-qlonglong | -qnolonglong
```

Purpose
Allows **long long** integer types in your program.

Default
The default with **xlC**, **xlc**, and **cc** is **-qlonglong**, which defines **_LONG_LONG** (**long long** types will work in programs). The default with **c89** is **-qnolonglong** (**long long** types are ignored).

Example
To compile myprogram.c so that **long long int**s are not allowed, enter:

```
xlC myprogram.c -qnolonglong
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

M

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-M
```

Purpose
Creates an output file that contains targets suitable for inclusion in a description file for the AIX **make** command.

Notes
The **-M** option is functionally identical to the "makedep" on page 182 option.

.u files are not **make** files; **.u** files must be edited before they can be used with the **make** command. For more information on this command, see *AIX Version 4 Commands Reference*.

If you do not specify the "o" on page 190 option, the output file generated by the **-M** option is created in the current directory. It has a **.u** suffix. For example, the command:

```
xlC -M person_years.c
```

produces the output file person_years.u.

A **.u** file is created for every input file with a **.c** or **.i** suffix. Output **.u** files are not created for any other files. For example, the command:

```
xlC -M conversion.c filter.c /lib/libm.a
```

produces two output files, **conversion.u** and **filter.u** (and an executable file as well). No **.u** file is created for the library.

If the current directory is not writable, no **.u** file is created. If you specify **-o***file_name* along with **-M**, the **.u** file is placed in the directory implied by "o" on page 190*file_name*. For example, for the following invocation:

```
xlC -M -c t.c -o /tmp/t.o
```

places the **.u** output file in **/tmp/t.u**.

Format of the 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, described in Directory Search Sequence for Include Files Using Relative Path Names. (If the include file is not found, it is not added to the **.u** file.)

Files with no include statements produce output files containing one line that lists only the input file name.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Preprocessor Options" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

ma

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-ma
```

Purpose
Substitutes inline code for calls to function **alloca** as if **#pragma alloca** directives are in the source code.

Notes
If **#pragma alloca** is unspecified, or if you do not use **-ma**, **alloca** is treated as a user-defined identifier rather than as a built-in function.

C++ programs must also specify **#include <malloc.h>** to include the **alloca** function declaration.

Example
To compile myprogram.c so that calls to the function **alloca** are treated as inline, enter:

```
xlC myprogram.c -ma
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

macpstr

Option Type	Default Value	#pragma options	C	C++

-q*option*	nomacpstr	MACPSTR	x	

Syntax

```
-qmacpstr | -qnomacpstr
MACPSTR | NOMACPSTR
```

Purpose
Converts Pascal string literals into null-terminated strings where the first byte contains the length of the string.

Notes
A Pascal string literal always contains the characters **"\p**. The characters **\p** in the middle of a string do not form a Pascal string literal; the characters must be *immediately preceded* by the " (double quote) character.

The final length of the Pascal string literal can be no longer than 255 bytes (the maximum length that can fit in a byte).

For example, the **-qmacpstr** converts:

```
"\pABC"
```

to:

```
'\03' , 'A' , 'B' , 'C' , '\0'
```

The compiler ignores the **-qmacpstr** option when the "mbcs, dbcs" on page 184 or "mbcs, dbcs" on page 184 option is active because Pascal-string-literal processing is only valid for one-byte characters.

The **#pragma options** keyword **MACPSTR** is only valid at the top of a source file before any C or C++ source statements. If you attempt to use it in the middle of a source file, it is ignored and the compiler issues an error message.

Examples of Pascal String Literals
The compiler replaces trigraph sequences by the corresponding single-character representation. For example:

```
"??/p pascal string"
```

becomes:

```
"\p pascal string"
```

The following are examples of valid Pascal string literals:

ANSI Mode

```
"\p pascal string"
```

Each instance of a new-line character and an immediately preceding backslash (\) character is deleted, splicing the physical source lines into logical ones. For example:

```
"\p pascal \
 string"
```

Two Pascal string literals are concatenated to form one Pascal string literal. For example:

```
"\p ABC" "\p DEF"
```

or

```
"\p ABC" "DEF"
```

becomes:

```
"\06ABCDEF"
```

For the macro ADDQUOTES:

```
#define ADDQUOTES (x) #x
```

where **x** is:

```
\p pascal string
```

or

```
\p pascal \
 string
```

becomes:

```
"\p pascal string"
```

Note however that:

```
ADDQUOTES(This is not a "\p pascal string")
```

becomes:

```
"This is not a \"\\p pascal string\""
```

Extended Mode

Is the same as ANSI mode, except the macro definition would be:

```
#define ADDQUOTES_Ext (x) "x"
```

Where x is the same as in the ANSI example:

```
\p pascal string
\p pascal \
 string
```

String Literal Processing

The following describes how Pascal string literals are processed.

- Concatenating a Pascal string literal to a normal string gives a non-Pascal string. For example:

  ```
  "ABC" "\pDEF"
  ```

  gives:

  ```
  "ABCpDEF"
  ```

- A Pascal string literal cannot be concatenated with a **wide** string literal.
- The compiler truncates a Pascal string literal that is longer than 255 bytes (excluding the length byte and the terminating NULL) to 255 characters.
- The compiler ignores the **-qmacpstr** option if "mbcs, dbcs" on page 184 or "mbcs, dbcs" on page 184 is used, and issues a warning message.
- Because there is no Pascal-string-literal processing of wide strings, using the escape sequence **\p** in a wide string literal with the **-qmacpstr** option, generates a warning message and the escape sequence is ignored.
- The Pascal string literal is *not* a basic type different from other C or C++ string literals. After the processing of the Pascal string literal is complete, the resulting string is treated the same as all other strings. If the program passes a C string to a function that expects a Pascal string, or vice versa, the behavior is undefined.
- Concatenating two Pascal string literals, for example, **strcat()**, does not result in a Pascal string literal. However, as described above, two adjacent Pascal string literals can be concatenated to form one Pascal string literal in which the first byte is the length of the new string literal.
- Modifying any byte of the Pascal string literal after the processing has been completed does not alter the original length value in the first byte.
- No errors or warnings are issued when the bytes of the processed Pascal string literal are modified.
- Entering the characters:

  ```
  '\p' , 'A' , 'B' , 'C' , '\0'
  ```

  into a character array does not form a Pascal string literal.

Example
To compile mypascal.c and convert string literals into null-terminated strings, enter:

```
xlC mypascal.c -qmacpstr
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

maf

Option Type	Default Value	#pragma options	C	C++
-q*option*	maf	MAF	x	x

Syntax

```
-qmaf | -qnomaf
MAF | NOMAF
```

Purpose
Specifies whether floating-point multiply-add instructions are to be generated. This option affects the precision of floating-point intermediate results. Before using this option, see RISC System/6000 Floating Point Hardware for more information about floating-point operations.

Notes
This option is obsolete. Use "float" on page 127**=maf** in your new applications.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

makedep

Option Type	Default Value	#pragma options	C	C++
-q*option*	-	-	x	x

Syntax

```
-qmakedep
```

Purpose
Creates an output file that contains targets suitable for inclusion in a description file for the AIX **make** command.

Notes
The **-qmakedep** option is functionally identical to the "M" on page 177 option.

.u files are not **make** files; **.u** files must be edited before they can be used with the **make** command. For more information on this command, see *AIX Version 4 Commands Reference*.

If you do not specify the "o" on page 190 option, the output file generated by the **-qmakedep** option is created in the current directory. It has a **.u** suffix. For example, the command:

```
    xlC -qmakedep person_years.c
```

produces the output file person_years.u.

A **.u** file is created for every input file with a **.c** or **.i** suffix. Output **.u** files are not created for any other files. For example, the command:

```
    xlC -qmakedep conversion.c filter.c /lib/libm.a
```

produces two output files, **conversion.u** and **filter.u** (and an executable file as well). No **.u** file is created for the library.

If the current directory is not writable, no **.u** file is created. If you specify **-o***file_name* along with **-qmakedep**, the **.u** file is placed in the directory implied by "o" on page 190*file_name*. For example, for the following invocation:

```
    xlC -qmakedep -c t.c -o /tmp/t.o
```

places the **.u** output file in **/tmp/t.u**.

Format of the 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, described in Directory Search Sequence for Include Files Using Relative Path Names. (If the include file is not found, it is not added to the **.u** file.)

Files with no include statements produce output files containing one line that lists only the input file name.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Preprocessor Options" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

maxmem

Option Type	Default Value	#pragma options	C	C++
-q*option*	▶ C maxmem=2048 ▶ C++ maxmem=8192	-	x	x

Syntax

```
-qmaxmem=size
```

Purpose
Limits the amount of memory used for local tables of specific, memory-intensive optimizations to *size* kilobytes. If that memory is insufficient for a particular optimization, the scope of the optimization is reduced.

Notes

- A *size* value of -1 permits each optimization to take as much memory as it needs without checking for limits. Depending on the source file being compiled, the size of subprograms in the source, the machine configuration, and the workload on the system, this might exceed available system resources.
- The limit set by **maxmem** is the amount of memory for specific optimizations, and not for the compiler as a whole. Tables required during the entire compilation process are not affected by or included in this limit.
- Setting a large limit has no negative effect on the compilation of source files when the compiler needs less memory.
- Limiting the scope of optimization does not necessarily mean that the resulting program will be slower, only that the compiler may finish before finding all opportunities to increase performance.
- Increasing the limit does not necessarily mean that the resulting program will be faster, only that the compiler is better able to find opportunities to increase performance if they exist.

Depending on the source file being compiled, the size of the subprograms in the source, the machine configuration, and the workload on the system, setting the limit too high might lead to page-space exhaustion. In particular, specifying **-qmaxmem=-1** allows the compiler to try and use an infinite amount of storage, which in the worst case can exhaust the resources of even the most well-equipped machine.

Example
To compile myprogram.c so that the memory specified for local table is **16384** kilobytes, enter:

```
xlC myprogram.c -qmaxmem=16384
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

mbcs, dbcs

Option Type	Default Value	#pragma options	C	C++
-q*option*	nombcs	DBCS	x	x

Syntax:

```
-qmbcs | -qdbcs | -qnombcs | -qnodbcs
MBCS | DBCS | NOMBCS | NODBCS
```

Purpose
Use the **-qmbcs** option if your program contains multibyte characters. The **-qmbcs** option is equivalent to **-qdbcs**.

Notes
Multibyte characters are used in certain languages such as Japanese and Korean.

Example
To compile myprogram.c if it contains multibyte characters, enter:

```
xlC myprogram.c -qmbcs
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
"National Languages Support in VisualAge C++" on page 47
"Multibyte Character Support" on page 48
Equivalent Batch Compile-Link and Incremental Build Options

mkshrobj

Option Type	Default Value	#pragma options	C	C++
-q*option*	-	-	x	x

Syntax

```
-qmkshrobj[=priority]
```

Purpose
Creates a shared object from generated object files. This option, together with the related options described below, should be used instead of the **makeC++SharedLib** command. The advantage to using this option is that the compiler will automatically include and compile the template instantiations in the tempinc directory.

The compiler will automatically export all global symbols from the shared object unless one explicitly specifies which symbols to export with the -bE:, -bexport: or -bexpall options.

The priority suboption has no effect if the you use the **xlc** command to link with or the shared object has no static initialization.

Suboption

priority	Specifies the priority level for the file. *priority* may be any number from -214782623 (highest priority-initialized first) to 214783647 (lowest priority-initialized last). Numbers from -214783648 to -214782624 are reserved for system use. If no priority is specified the default priority of 0 is used. The priority is not used when linking shared objects (using the **xlc** command) written in C.

Related Options
The following xlC options are optionally used with **-qmkshrobj**

-o*shared_file.o*	Is the name of the file that will hold the shared file information. The default is `shr.o`.
-qexpfile=*filename*	Saves all exported symbols in *filename*. This option is ignored unless xlC automatically creates the export list.
-e *name*	Sets the entry name for the shared executable to name. The default is -bnoentry.

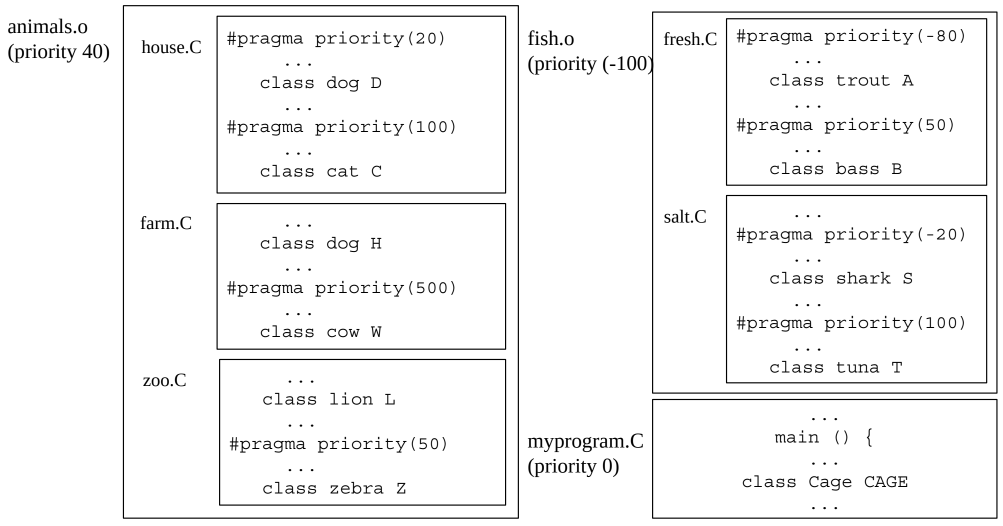
Notes
If you use -qmkshrobj to create a shared library, the compiler will:

1. If the user doesn't specify -bE:, -bexport:, -bexpall or -bnoexpall, create an export list containing all global symbols using the CreateExportList script. You can specify another script with the -tE/-B or -qpath=E: options.
2. If CreateExportList was used to create the export list and -qexpfile was specified, the export list is saved.
3. Calls the linker with the appropriate options and object files to build a shared object.

Example

The following example shows how to construct a shared library containing two shared objects using the the -qmkshrobj option, and the AIX **ar** command. The shared library is then linked with a file that contains the main function. Different priorities are used to ensure objects are initialized in the specified order.

The drawing below shows how the objects in this example are arranged in various files.



```
#pragma priority(20)
    ...
   class dog D
    ...
#pragma priority(100)
    ...
   class cat C
```

```
    ...
   class dog H
    ...
#pragma priority(500)
    ...
   class cow W
```

```
    ...
   class lion L
    ...
#pragma priority(50)
    ...
   class zebra Z
```

```
#pragma priority(-80)
    ...
   class trout A
    ...
#pragma priority(50)
    ...
   class bass B
```

```
    ...
#pragma priority(-20)
    ...
   class shark S
    ...
#pragma priority(100)
    ...
   class tuna T
```

```
    ...
  main () {
    ...
class Cage CAGE
    ...
```

The first part of this example shows how to use the **-qpriority=***N* option and the **#pragma priority**(*N*) directive to specify the initialization order for objects within the object files.

The example shows how to make two shared objects: animals.o containing object files compiled from house.C, farm.C, and zoo.C, and fish.o containing object files compiled from fresh.C and salt.C. The -**qmkshrobj**=*P* option is used to specify the priority of the initialization of the shared objects.

The priority values for the shared objects are chosen so that all the objects in fish.o are initialized before the objects in myprogram.o, and all the objects in animals.o are initialized after the objects in myprogram.o.

To specify this initialization order, follow these steps:

1. Develop an initialization order for the objects in house.C, farm.C, and zoo.C:
   a. To ensure that the object lion L in zoo.C is initialized before any other objects in either of the other two files, compile zoo.C using a **-qpriority=***N* option with *N* less than zero so both objects have a priority number less than any other objects in farm.C and house.C:
      ```
      xlC zoo.C -c -qpriority=-50
      ```
   b. Compile the house.C and farm.C files without specifying the **-qpriority=***N* option (so *N*=0) so objects within the files retain the priority numbers specified by their **#pragma priority**(*N*) directives:
      ```
      xlC house.C farm.C -c
      ```
   c. Combine these three files in a shared library. Use **xlC -qmkshrobj** to construct a library animals.o with a priority of 40:
      ```
      xlC -qmkshrobj=40 -o animals.o house.o farm.o zoo.o
      ```
2. Develop an initialization order for the objects in fresh.C, and salt.C:
   a. Compile the fresh.C and salt.C files:
      ```
      xlC fresh.C salt.C -c
      ```
   b. To assure that all objects in fresh.C and salt.C are initialized before any other objects, use **xlC -qmkshrobj** to construct a library fish.o with a priority of -100.
      ```
      xlC -qmkshrobj=-100 -o fish.o fresh.o salt.o
      ```

Because the shared library fish.o has a lower priority number (-100) than animals.o (40), when the files are placed in an archive file with the **ar** command, their objects are initialized first.

3. Compile myprogram.C that contains the function main to produce an object file myprogram.o. By not specifying a priority, this file is compiled with a default priority of zero, and the objects in main have a priority of zero.

   ```
   xlC myprogram.C -c
   ```

4. To create a library that contains the two shared objects animals.o and fish.o, you use the **ar** command. To produce an archive file, libzoo.a, enter the command:

   ```
   ar rv libzoo.a animals.o fish.o
   ```

   where:

rv	Are two **ar** options. r replaces a named file if it already appears in the library, and v writes to standard output a file-by-file description of the making of the new library.
libzoo.a	Is the name you specified for the archive file that will contain the shared object files and their priority levels.
animals.o fish.o	Are the two shared files you created with **xlC -qmkshrobj**.

5. To produce an executable file, animal_time, so that the objects are initialized in the order you have specified, enter:

   ```
   xlC -oanimal_time myprogram.o -L. -lzoo
   ```

6. The order of initialization of the objects is shown in the following table.

Order of Initialization of Objects in libzoo.a

File	Class Object	Priority Value	Comment
"fish.o"		-100	All objects in "fish.o" are initialized first because they are in a library prepared with **-qmkshrobj=-100** (lowest priority number, -100, specified for any files in this compilation)
	"shark S"	-100(-200)	Initialized first in "fish.o" because within file, #pragma priority(-200)
	"trout A"	-100(-80)	#pragma priority(-80)
	"tuna T"	-100(10)	#pragma priority(10)
	"bass B"	-100(500)	#pragma priority(500)

Order of Initialization of Objects in libzoo.a			
"myprog.o"		0	File generated with no priority specifications; default is 0
	"CAGE"	0(0)	Object generated in **main** with no priority specifications; default is 0
"animals.o"		40	File generated with **-qmkshrobj=40**
	"lion L"	40(-50)	Initialized first in file "animals.o" compiled with -qpriority=-50
	"horse H"	40(0)	Follows with priority of 0 (since -qpriority=*N* not specified at compilation and no #pragma priority(*N*) directive)
	"dog D"	40(20)	Next priority number (specified by #pragma priority(20))
	"zebra N"	40(50)	Next priority number from #pragma priority(50)
	"cat C"	40(100)	Next priority number from #pragma priority(100)
	"cow W"	40(500)	Next priority number from #pragma priority(500) (Initialized last)

You can place both nonshared and shared files with different priority levels in the same archive library using the AIX **ar** command.

RELATED CONCEPTS

"Constructing a Library" on page 69

RELATED TASKS

"Initialize Shared Library (C++)" on page 70

RELATED REFERENCES

"CreateExportList Command" on page 77
"makeC++SharedLib Command" on page 79
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

namemangling (C++)

Option Type	Default Value	#pragma options	C	C++
-q*option*	namemangling=ansi	-		x

Syntax

```
-qnamemangling=ansi|compat
```

Purpose
Choses the name mangling scheme for external symbol names generated from C++ source code. You can select one of two mangling schemes:

- **compat**
  This scheme uses the same mangling scheme as earlier versions of VisualAge C++ or IBM C and C++ Compilers. Use this scheme for compatibility with link modules built with ealier versions of the compiler.
- **ansi**
  This scheme complies with the C++ standard. Use this scheme for compatibility with standard C++.

By default VisualAge C++ uses a scheme that conforms to specifications of the C++ standard.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Linkage Options" on page 42
Equivalent Batch Compile-Link and Incremental Build Options

noprint

Option Type	Default Value	#pragma options	C	C++
-q*option*	-	-	x	x

Syntax

```
-qnoprint
```

Purpose
Suppresses listings. **-qnoprint** overrides all of the listing-producing options, regardless of where they are specified.

Notes
The default is not to suppress listings if they are requested.

The options that produce listings are:
- "attr" on page 99
- "list" on page 175
- -qlistopt
- "source" on page 217
- "xref" on page 241

Example
To compile myprogram.c and suppress all listings, even if some files have **#pragma options source** and similar directives, enter:

```
xlC myprogram.c -qnoprint
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

o

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-o file_spec
```

Purpose
Specifies an output location for the object, assembler, or executable files created by the compiler. When the **-o** option is used during compiler invocation, *file_spec* can be the name of either a file or a directory. When the **-o** option is used during direct linkage-editor invocation, *file_spec* can only be the name of a file.

Notes
When **-o** is specified as part of a complier invocation, *file_spec* can be the relative or absolute path name of either a directory or a file.

1. If *file_spec* is the name of a directory, files created by the compiler are placed into that directory.
2. If a directory with the name *file_spec* does not exist, the **-o** option specifies that the name of the file produced by the compiler will be *file_spec*. Otherwise, files created by the compiler will take on their default names. For example, the following compiler invocation:
   `xlC test.c -c -o new.o`
   produces the object file **new.o** instead of **test.o** , and
   `xlC test.c -o new`
   produces the object file **new** instead of **a.out**
   A *file_spec* with a C or C++ source file suffix (**.C**, **.c**, or **.i**), such as my_text.c or bob.i, results in an error and neither the compiler nor the linkage editor is invoked.
   If you "c" on page 103 and **-o** together and the *filespec* does not specify a directory, you can only compile one source file at a time. In this case, if more than one source file name is listed in the compiler invocation, the compiler issues a warning message and ignores **-o**.

The "E" on page 116, "P" on page 195, and "syntaxonly (C Only)" on page 225 options override the **-o***filename* option.

Example

1. To compile myprogram.c so that the resulting file is called **myaccount**, assuming that no directory with name **myaccount** exists, enter:

   ```
   xlC myprogram.c -o myaccount
   ```

If the directory **myaccount** does exist, the executable file produced by the compiler is placed in the **myaccount** directory.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Linkage Options" on page 42
Equivalent Batch Compile-Link and Incremental Build Options

objmodel

Option Type	Default Value	#pragma options	C	C++
-q*option*	=compat	#pragma object_model(*option*)		x

Syntax

```
    -qobjmodel=compat | ibm
#pragma object_model(compat|ibm|pop)
```

Purpose
Sets the type of object model.

Notes
Usage modes for **objmodel** are:

`-qobjmodel=compat`	Uses the xlC object model compatible with previous versions of the compiler.
`-qobjmodel=ibm`	Uses the new object model.

Example
To compile myprogram.C with the ibm object model, enter:

```
xlC myprogram.C -qobjmodel=ibm
```

RELATED CONCEPTS

Object Models

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

once (C Only)

Option Type	Default Value	#pragma options	C	C++
-q*option*	noonce	ONCE	x	

Syntax

```
-qonce | -qnoonce
ONCE | NOONCE
```

Purpose
Avoids including a header file more than once even if it is specified in several of the files you are compiling.

Notes
The compiler uses the full path name to determine if a file has already been included. No attempt is made to resolve **.** or **..** in the path name. **#include** statements that include **.** or **..** in the path statements may cause the same file to be included more than once.

The **#pragma options** keyword **ONCE** may appear anywhere in your code. It can be turned on and off by specifying **ONCE** and **NOONCE**, respectively.

Important!
Do not use the **-qonce** option if *both* of the following conditions are true: 1. You include both **stdio.h** and **stdarg.h** (in that order) in your source files, and, 2. You are using the macro **va_list**. **va_list** must be defined twice to have any effect, and **-qonce** defeats this purpose.

Example
The following example shows how the compiler resolves whether a file has already been included.

```
#include <stdio.h>         /* Found in /usr/include/stdio.h */
#include <stdio.h>         /* Already included              */
#include </usr/include/stdio.h>    /* Already included      */
#include <./stdio.h>  /* Resolves to /usr/include/./stdio.h */
                      /* which is the same file, but this   */
                      /* file will be included again.       */
```

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

O, optimize

Option Type	Default Value	#pragma options	C	C++
-q*option* *-flag*	nooptimize	-	x	all except: -O4,-O5 -qoptimize=4,5 OPTimize=4,5

Syntax

```
-O | -O2 | -O3 | -O4 | -qoptimize | -qoptimize=2 | -qoptimize=3 |
      -qoptimize=4 | -qoptimize=5 | -qnooptimize | -qoptimize=0
```

```
OPTimize | OPTimize=2 | OPTimize=3 | OPTimize=4 | OPTimize=5 |
      NOOPTimize | OPTimize=0
```

Purpose
Optimizes code at a choice of levels during compilation.

Notes
You can abbreviate **-qoptimize...** to **-qopt...**. For example, **-qnoopt** is equivalent to **-qnooptimize**.

Increasing the level of optimization may or may not result in additional performance improvements, depending on whether additional analysis detects further opportunities for optimization.

Compilations with optimizations may require more time and machine resources than other compilations.

Optimization can cause statements to be moved or deleted, and generally should not be specified along with the "g" on page 135 flag for debugging programs. The debugging information produced may not be accurate.

The levels of optimization are:

-qNOOPTimize	(Same as **-qOPTimize=0**.) Performs only quick local optimizations such as constant folding and elimination of local common subexpressions. This setting implies **-qstrict_induction** unless **-qnostrict_induction** is explicitly specified.
-O, -qOPTimize	Performs optimizations that the compiler developers considered the best combination for compilation speed and runtime performance. The optimizations may change from product release to release. If you need a specific level of optimization, specify the appropriate numeric value. This setting implies **-qstrict_induction** unless **-qnostrict_induction** is explicitly specified.
-O2, -qOPTimize=2	Same as **-O**.

-O3, -qOPTimize=3	Performs additional optimizations that are memory intensive, compile-time intensive, or both. These optimizations are performed in addition to those performed with only the **-O** option specified. They are recommended when the desire for runtime improvement outweighs the concern for minimizing compilation resources. This level is the compiler's highest and most aggressive level of optimization. **-O3** performs optimizations that have the potential to slightly alter the semantics of your program. It also applies the **-O2** level of optimization with unbounded time and memory. The compiler guards against these optimizations at **-O2**. You can use the "strict" on page 221 option with **-O3** to turn off the aggressive optimizations that might change the semantics of a program. **-qstrict** combined with **-O3** invokes all the optimizations performed at **-O2** as well as further loop optimizations. Note that the **-qstrict** compiler option must appear after the **-O3** option, otherwise it is ignored. The aggressive optimizations performed when you specify **-O3** are: 1. Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed. Loads and floating-point computations fall into this category. This optimization is aggressive because it may place such instructions onto execution paths where they *will* be executed when they *may* not have been according to the actual semantics of the program. For example, a loop-invariant floating-point computation that is found on some, but not all, paths through a loop will not be moved at **-O2** because the computation may cause an exception. At **-O3**, the compiler will move it because it is not certain to cause an exception. The same is true for motion of loads. Although a load through a pointer is never moved, loads off the static or stack base register are considered movable at **-O3**. Loads in general are not considered to be absolutely safe at **-O2** because a program can contain a declaration of a static array `a` of 10 elements and load `a[60000000003]`, which could cause a segmentation violation. The same concepts apply to scheduling. **Example**: In the following example, at **-O2**, the computation of `b+c` is not moved out of the loop for two reasons: a. it is considered dangerous because it is a floating-point operation it does not occur on every path

-O4, -qOPTimize=4	Valid only for C program compilations. This option is the same as **-O3**, except that it also: • Sets the "ipa (C Only)" on page 151 option • Sets the "arch" on page 96 and "tune" on page 230 options to the architecture of the compiling machine **Note:** Later settings of **-O**, "cache" on page 104 "ipa (C Only)" on page 151, "arch" on page 96, and "tune" on page 230 options will override the settings implied by the **-O4** option.
▶ C **-O5, -qOPTimize=5**	This option is the same as **-O4**, except that it: • Sets the **-qipa=level=2** option to perform full interprocedural data flow and alias analysis. **Note:** Later settings of **-O**, **-qcache**, **-qipa**, **-qarch**, and **-qtune** options will override the settings implied by the **-O5** option.

Example

To compile myprogram.c for maximum optimization, enter:

```
xlC myprogram.c -O3
```

RELATED CONCEPTS

Overview of Optimization

RELATED TASKS

Optimize Your Program

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

P

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-P
```

Purpose

Preprocesses the C or C++ source files named in the compiler invocation and creates an output preprocessed source file, *file_name*.**i** for each input source file, *file_name*.**c** or *file_name*.**C**. The **-P** option calls the preprocessor directly as **/usr/vac/exe/xlCcpp**.

Notes
The **-P** option retains all white space including line-feed characters, with the following exceptions:

- All comments are reduced to a single space (unless “C” on page 103 is specified).
- Line feeds at the end of preprocessing directives are not retained.
- White space surrounding arguments to function-style macros is not retained.

#linedirectives are not issued.

The **-P** option cannot accept a preprocessed source file, *file_name*.**i**as input. Source files with unrecognized filename suffixes are treated and preprocessed as C files, and no error message is generated.

In extended mode, the preprocessor interprets the backslash character when it is followed by a new-line character as line-continuation in:

- macro replacement text
- macro arguments
- comments that are on the same line as a preprocessor directive.

Line continuations elsewhere are processed in **ANSI** mode only.

The **-P** option is overridden by the “E” on page 116 option. The **-P** option overrides the “c” on page 103, “o” on page 190**,** and “syntaxonly (C Only)” on page 225 option. The “C” on page 103 option may used in conjunction with both the **-E** and **-P** options.

The default is to compile and link-edit C or C++ source files to produce an executable file.

RELATED REFERENCES
“List of Batch Compiler Options and Their Defaults” on page 24
“Options that Specify Compiler Output” on page 40
Equivalent Batch Compile-Link and Incremental Build Options

p

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-p
```

Purpose
Sets up the object files produced by the compiler for profiling.

If the **-qtbtable** option is not set, the **-p** option will generate full traceback tables.

Note: When compiling and linking in separate steps, the **-p** option must be specified in both steps.

Example
To compile myprogram.c so that it can be used with the AIX **prof** command, enter:

```
xlC myprogram.c -p
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
"pg" on page 201 Compiler Option
Equivalent Batch Compile-Link and Incremental Build Options
prof command in the *AIX Version 4 Commands Reference*, for details on profiling.

pascal

Option Type	Default Value	#pragma options	C	C++
-q*option*	nopascal	-	x	

Syntax

```
-qpascal | -qnopascal
```

Purpose
Ignores the word **pascal** in type specifiers and function declarations.

Notes
This option can be used to improve compatibility of IBM VisualAge C++ programs on some other systems.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

path

Option Type	Default Value	#pragma options	C	C++
-*q*path	-	-	x	x

Syntax

```
-qpath=components:path
```

Purpose
Constructs alternate program names. The programs specified by *components* and located in the *path* directory are used in place of the regular programs. *components* can be a combination of any of the following:

Program	**Description**
c	Compiler front end
b	Compiler back end
i	Compiler inliner
p	Compiler preprocessor
a	Assembler
I	Interprocedural Analysis tool - compile phase
L	Interprocedural Analysis tool - link phase
l	Linkage editor
E	CreateExportList utility
m	Linkage helper (**munch** utility)

The **-qpath** option overrides the “F” on page 124*config_file*, “t” on page 227, and “B” on page 99 options.

Example
To compile myprogram.c using a substitute **xlC** compiler in **/lib/tmp/mine/** enter:

```
xlC myprogram.c -qpath=c:/lib/tmp/mine/
```

To compile myprogram.c using a substitute linkage editor in **/lib/tmp/mine/**, enter:

```
xlC myprogram.c -qpath=l:/lib/tmp/mine/
```

RELATED REFERENCES

“List of Batch Compiler Options and Their Defaults” on page 24
“Options that Specify Compiler Characteristics” on page 38
Equivalent Batch Compile-Link and Incremental Build Options

pdf1, pdf2

Option Type	Default Value	#pragma options	C	C++
-q*option*	nopdf1 nopdf2	-	x	x

Syntax

```
-qpdf1 | -qpdf2 | -qnopdf1 | -qnopdf2
```

Purpose
Tunes optimizations through Profile-Directed Feedback (PDF), where results from one or more sample program executions are used to improve optimization near conditional branches and in frequently executed code sections.

Notes
To use PDF:

1. Compile some or all of the source files in a program with the **-qpdf1** option. **main** must be compiled. The **“l” on page 160pdf** option is required during the link step, the “O, optimize” on page 192 option is recommended for optimization. Pay special attention to the compiler options used to compile the files, because you will need to use the same options later.
2. Run the program all the way through, using a typical data set. The program records profiling information when it finishes. You can run the program multiple times with different data sets, and the profiling information is accumulated to provide an accurate count of how often branches are taken and blocks of code are executed.

   Important: Use data that is representative of the data that will be used during a normal run of your finished program.
3. Recompile your program, using the same compiler options as before but changing **-qpdf1** to **-qpdf2**. Remember that “L” on page 159, “l” on page 160, and some others are linker options, and you can change them at this point. In particular, leave the **-lpdf** option out. In this second compilation, the accumulated profiling information is used to fine-tune the optimizations. The resulting program contains no profiling overhead and runs at full speed.

For optimum performance, use the “O, optimize” on page 192 option with all compilations when you use PDF (as in the example above). With “O, optimize” on page 192 page 192 optimization, one of the most important PDF optimizations (moving code before branches to fill delay slots) is not done.

The profile is placed in the current working directory, or the directory named by the **PDFDIR** environment variable if that variable is set.

To avoid wasting compilation and execution time, make sure the **PDFDIR** environment variable is set to an absolute path; otherwise, you might run the application from the wrong directory so that it cannot locate the profile data files. If that happens, the program may not be optimized correctly or may be stopped by a segmentation fault. A segmentation fault might also happen if you change the value of the **PDFDIR** variable and execute the application before finishing the PDF process.

Because this option requires compiling the entire application twice, it is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production.

Restrictions

- Do not mix PDF files created by the current version of VisualAge C++ with PDF files created by previous versions.
- PDF optimizations also require at least level 2 of “O, optimize” on page 192.
- The main program must be compiled with PDF for profiling to work properly. If you want to use this option to optimize a library or other code that does not usually incorporate a main program, supply a main program for the first PDF compilation, then omit the main program for the second PDF compilation.
- Do not compile or run two different applications that use the same **PDFDIR** directory at the same time.
- You must use the same set of compiler options at all compilation steps for a particular program; otherwise, PDF cannot optimize your program correctly, and may even slow it down. All compiler settings must be the same, including any supplied by configuration files.
- If you do compile a program with **-qpdf1**, remember that it will generate profiling information when it runs, which involves some performance overhead. This overhead goes away when you recompile with **-qpdf2** or with no PDF at all.

The following commands are available for managing the **PDFDIR** directory:

`resetpdf [pathname]`	Zeros out all profiling information (but does not remove the data files) from the pathname directory; or if pathname is not specified, from the **PDFDIR** directory; or if **PDFDIR** is not set, from the current directory. When you make changes to the application and recompile some files, the profiling information for those files is automatically reset, because the changes may alter the program flow. Run **resetpdf** to reset the profiling information for the entire application, after making significant changes that may affect execution counts for parts of the program that were not recompiled.

`cleanpdf [pathname]`	Removes all profiling information from the pathname directory; or if pathname is not specified, from the **PDFDIR** directory; or if **PDFDIR** is not set, from the current directory. Removing the profiling information reduces the runtime overhead if you change the program and then go through the PDF process again. Run this program after compiling with **-qpdf2**, or after finishing with the PDF process for a particular application. If you continue using PDF with an application after running **cleanpdf**, you must recompile all the files with **-qpdf1**.

Example 1
Here are the steps for a simple example:

1. first, set the **PDFDIR** environment variable:
   ```
   export PDFDIR=/home/user
   ```
2. Compile all files with **-qpdf1** and "O, optimize" on page 192, and link with **-lpdf**.
   ```
   xlC -qpdf1 -lpdf -O3 file1.c file2.c file3.c -L/usr/vacpp/lib
   ```
3. Run with one set of input data:
   ```
   a.out < sample.data
   ```
4. Recompile all files with **-qpdf2** and "O, optimize" on page 192:
   ```
   xlC -qpdf2 -O3 file1.c file2.c file3.c
   ```

The program should now run faster than without PDF, if the sample used data was typical of actual program data.

Note: When using **-qpdf1**, specify the search location for its libraries with the "L" on page 159 compiler option, as shown in step 2 above.

Example 2
Here are the steps for a more elaborate example.

1. Set the **PDFDIR** environment variable:
   ```
   export PDFDIR=/home/user
   ```
2. Compile most of the files with **-qpdf1**.
   ```
   xlC -qpdf1 -O3 -c file1.c file2.c file3.c -L/usr/vacpp/lib
   ```
3. This file is not so important to optimize:
   ```
   xlC -c file4.c
   ```
4. Non-PDF object files like `file4.o` can be linked in:
   ```
   xlC -qpdf1 -lpdf file1.o file2.o file3.o file4.o -L/usr/vacpp/lib
   ```
5. Run several times with different input data:
   ```
   a.out < polar_orbit.data
   a.out < elliptical_orbit.data
   a.out < geosynchronous_orbit.data
   ```
6. You do not need to recompile the source of non-PDF object files:
   ```
   xlC -qpdf2 -O3 file1.c file2.c file3.c
   ```
7. Link all the object files into the final application:
   ```
   xlC file1.o file2.o file3.o file4.o
   ```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

pg

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-pg
```

Purpose
Sets up the object files for profiling, but provides more information than is provided by the "p" on page 196 option.

If the **-qtbtable** option is not set, the **-pg** option will generate full traceback tables.

Example
To compile myprogram.c for use with the AIX **gprof** command, enter:

```
xlC myprogram.c -pg
```

Remember to compile *and* link with the **-pg** option. For example:

```
xlC myprogram.c -pg -c
xlC myprogram.o -pg -o program
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
Equivalent Batch Compile-Link and Incremental Build Options
gprof command in the *AIX Version 4 Commands Reference*, for details on profiling.

phsinfo

Option Type	Default Value	#pragma options	C	C++
-q*option*	nophsinfo	-	x	x

Syntax

```
-qphsinfo | -qnophsinfo
```

Purpose
Reports the time taken in each compilation phase. Phase information is sent to standard output.

The output takes the form *number1* | *number2* for each phase where *number1* represents the CPU time used by the compiler and *number2* represents the total of the compiler time and the time that the CPU spends handling system calls.

Example
To compile myprogram.c and report the time taken for each phase of the compilation, enter:

```
xlC myprogram.C -qphsinfo
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

priority

Option Type	Default Value	#pragma options	C	C++
-flag	-	PRORITY=*number*		x

Syntax

```
-qpriority=number
PRIORITY=number
```

Purpose
Specifies the priority level for the initialization of static constructors

Notes:

number	Is the initialization priority level assigned to the static constructors within a file, or the priority level of a shared or non-shared file or library. You can specify a priority level from -(2147483647 + 1) (highest priority) to +2147483647 (lowest priority).

Example
To compile the file myprogram.C to produce an object file myprogram.o so that objects within that file have an initialization priority of -200, enter:

```
xlC myprogram.C -c -qpriority=-200
```

All objects in the resulting object file will be given an initialization priority of -200, provided that the source file contains no **#pragma priority(***number***)** directives specifying a different priority level.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

proclocal, procimported, procunknown

Option Type	Default Value	#pragma options	C	C++
-q*option*	proclocal*	PROCLOCal, PROCIMPorted, PROCUNKnown	x	x

Syntax

```
    -qproclocal | -qproclocal=names
    -qprocimported | -qprocimported=names
    -qprocunknown | -qprocunknown=names
    PROCLOCAL | PROCLOCAL=names
    PROCIMPORTED | PROCIMPORTED=names
    PROCUNKNOWN | PROCUNKNOWN=names
```

Purpose
Marks functions as local, imported, or unknown.

Default
The default is to assume that all functions whose definition is in the current compilation unit are local (**proclocal**), and that all other functions are unknown (**procunknown**). If any functions that are marked as local resolve to shared library functions, the linkage editor will detect the error and issue warnings such as:

```
ld: 0711-768 WARNING: Object foo.o, section 1, function .printf:
          The branch at address 0x18 is not followed by a recognized no-op
          or TOC-reload instruction.  The unrecognized instruction is 0x83E1004C.
```

An executable file is produced, but it will not run. The error message indicates that a call to **printf** in object file **foo.o** caused the problem. When you have confirmed that the called routine should be imported from a shared object, recompile the source file that caused the warning and explicitly mark **printf** as imported. For example:

```
     xlC -c -qprocimported=printf foo.c
```

Notes

Local functions	Are statically bound with the functions that call them. **-qproclocal** changes the default to assume that all functions are local. **-qproclocal=***names* marks the named functions as local, where *names* is a list of function identifiers separated by colons (**:**). The default is not changed. Smaller, faster code is generated for calls to functions marked as local.

Imported functions	Are dynamically bound with a shared portion of a library. **-qprocimported** changes the default to assume that all functions are imported. **-qprocimported=***names* marks the named functions as imported, where *names* is a list of function identifiers separated by colons (**:**). The default is not changed. The code generated for calls to functions marked as imported might be larger, but it is faster than the default code sequence generated for functions marked as unknown. If any marked functions are resolved to statically bound objects, the generated code may be larger and run more slowly than the default code sequence generated for unknown functions.
Unknown functions	Are resolved to either statically or dynamically bound objects during link-editing. **-qprocunknown** changes the default to assume that all functions are unknown. **-qprocunknown=***names* marks the named functions as unknown, where *names* is a list of function identifiers separated by colons (**:**). The default is not changed.

Conflicts among the procedure-marking options are resolved in the following manner:

Options that list function names	The last explicit specification for a particular function name is used.
Options that change the default	This form does not specify a name list. The last option specified is the default for functions not explicitly listed in the name-list form.

Example

To compile myprogram.c along with the archive library **oldprogs.a** so that the functions **fun** and **sun** are specified as **local**, **moon** and **stars** are specified as **imported**, and **venus** is specified as **unknown**, enter:

```
xlC myprogram.c oldprogs.a -qprolocal=fun(int):sun()
  -qprocimported=moon():stars(float) -qprocunknown=venus()
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

profile

Option Type	Default Value	#pragma options	C	C++
-q*option*	noprofile	-	x	x

Syntax

```
-qprofile=ibm |-qprofile=p |-qprofile=pg |-qnoprofile
```

Purpose
Sets up the object files produced by the compiler for profiling. The suboption indicates the profiling tool.

If the -qtbtable option is not set, the -qprofile option will generate full traceback tables.

Notes
The **-qprofile** suboptions are:

`ibm`	Profiling information for use with the **Performance Analysis** tool in VisualAge C++.
`p`	Profiling information for use with the **prof** AIX command.
`pg`	Profiling information for use with the **gprof** AIX command.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
"p" on page 196
"pg" on page 201
Equivalent Batch Compile-Link and Incremental Build Options

proto

Option Type	Default Value	#pragma options	C	C++
-q*option*	noproto	PROTO	x	

Syntax

```
-qproto | -qnoproto
PROTO | NOPROTO
```

Purpose
Assumes all functions are prototyped.

Notes
This option asserts that procedure call points agree with their declarations even if the procedure has not been prototyped. Callers can pass floating-point arguments in floating-point registers only and not in General-Purpose Registers (GPRs). The compiler assumes that the arguments on procedure calls are the same types as the corresponding parameters of the procedure definition.

You can obtain warnings for functions that do not have prototypes.

Example
To compile my_c_program.c to assume that all functions are prototyped, enter:

```
xlC my_c_program.c -qproto
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"info" on page 144 Compiler Option
Equivalent Batch Compile-Link and Incremental Build Options

Q

Option Type	Default Value	#pragma options	C	C++
-flag	*See below.*	-	x	x

Syntax

```
-Q | -Q=threshold | -Q-names | -Q+names | -Q!
```

Purpose
In the C language, attempts to inline functions instead of generating calls to a function. Inlining is performed if possible, but, depending on which optimizations are performed, some functions might not be inlined.

In the C++ language, specifies which functions will be inlined instead of generating a call to a function.

Notes
The **-Q** option is functionally equivalent to the "inline" on page 148option.

Because inlining does not always improve run time, you should test the effects of this option on your code.

Do not attempt to inline recursive or mutually recursive functions.

Normally, application performance is optimized if you request optimization ("O, optimize" on page 192 option), and compiler performance is optimized if you do not request optimization.

The VisualAge C++ _inline, _Inline, and __inline C language keywords override all **-Q** options except **-Q!**. The compiler will try to inline functions marked with these keywords regardless of other **-Q** option settings.

To maximize inlining:
- for C programs, specify optimization ("O, optimize" on page 192) and also specify the appropriate **-Q** options for the C language.
- for C++ programs, specify optimization ("O, optimize" on page 192) but do not specify the **-Q** option.

In the C language, the following **-Q** options apply:

`-Q`	Attempts to inline all appropriate functions with 20 executable source statements or fewer, subject to the setting of any of the suboptions to the **-Q** option. If **-Q** is specified last, all functions are inlined.
`-Q!`	Does not inline any functions. If **-Q!** is specified last, no functions are inlined.

In the C++ language, the following **-Q** options apply:

`-Q`	Compiler inlines all functions that it can.
`-Q!`	Compiler does not inline any functions.

In the C language, the following **-Q** options apply:

`-Q=`*`threshold`*

Sets a size limit on the functions to be inlined. The number of executable statements must be less than or equal to *threshold* for the function to be inlined. *threshold* must be a positive integer. The default value is 20. Specifying a threshold value of **0** causes no functions to be inlined except those functions marked with the **__inline**, **_Inline**, or **_inline** keywords.

The *threshold* value applies to logical C statements. Declarations are not counted, as you can see in the example below:

```
increment()
{
 int a, b, i;
  for (i=0; i<10; i++) /* statement 1 */
  {
     a=i;              /* statement 2 */
     b=i;              /* statement 3 */
  }
}
```

`-Q-`*`names`*

Does not inline functions listed by *names*. Separate each *name* with a colon (:). All other appropriate functions are inlined. The option implies **-Q**.

For example:

```
-Q-salary:taxes:expenses:benefits
```

causes all functions except those named **salary**, **taxes**, **expenses**, or **benefits** to be inlined if possible.

A warning message is issued for functions that are not defined in the source file.

`-Q+`*`names`*	Attempts to inline the functions listed by *names* and any other appropriate functions. Each *name* must be separated by a colon (**:**). The option implies **-Q**.
	
	For example,
	`-Q+food:clothes:vacation`
	
	causes all functions named **food**, **clothes**, or **vacation** to be inlined if possible, along with any other functions eligible for inlining.
	
	A warning message is issued for functions that are not defined in the source file or that are defined but cannot be inlined.
	
	This suboption overrides any setting of the *threshold* value. You can use a threshold value of zero along with **-Q+***names* to inline specific functions. For example:
	`-Q=0`
	
	followed by:
	`-Q+salary:taxes:benefits`
	
	causes *only* the functions named **salary**, **taxes**, or **benefits** to be inlined, if possible, and no others.

Default
The default is to treat inline specifications as a hint to the compiler and depends on other options that you select:

- If you specify the "g" on page 135 option (to generate debug information), no functions are inlined.
- If you optimize your programs, (specify the "O, optimize" on page 192 option) the compiler attempts to inline the functions declared as inline.

Example
To compile the program myprogram.c so that no functions are inlined, enter:

```
xlC myprogram.c -O -Q!
```

To compile the program my_c_program.c so that the compiler attempts to inline functions of fewer than 12 lines, enter:

```
xlc my_c_program.c -O -Q=12
```

RELATED CONCEPTS

Overview of Optimization

RELATED TASKS

Optimize Your Application

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41

_inline, _Inline, and __inline
Equivalent Batch Compile-Link and Incremental Build Options

r

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-r
```

Purpose
Produces a relocatable object. This permits the output file to be produced even though it contains unresolved symbols.

Notes
A file produced with this flag is expected to be used as a file parameter in another call to **xlC**.

Example
To compile myprogram.c and myprog2.c into a single object file **mytest.o**, enter:

```
xlC myprogram.c myprog2.c -r -o mytest.o
```

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Linkage Options" on page 42
Equivalent Batch Compile-Link and Incremental Build Options

rndsngl

Option Type	Default Value	#pragma options	C	C++
-q*option*	norndsngl	RNDSNGL	x	x

Syntax:

```
-qrndsngl | -qnorndsngl
RNDSNGL | NORNDSNGL
```

Purpose
Specifies that the results of each single-precision (**float**) operation is to be rounded to single precision. **-qnorndsngl** specifies that rounding to single-precision happens only after full expressions have been evaluated.

Notes
This option is obsolete. Use "float" on page 127**=rndsngl**. in your new applications.

The "hsflt" on page 140 option overrides the **-qrndsngl** options.

The **-qrndsngl** option is intended for specific applications in which floating-point computations have known characteristics. Using this option when compiling other application programs can produce incorrect results without warning. See RISC System/6000 Floating Point Hardware before you use the **-qrndsngl** option.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

ro

Option Type	Default Value	#pragma options	C	C++
-q*option*	ro*	RO	x	x

Syntax:

```
-qro | -qnoro
RO | NORO
```

Purpose
Specifies the storage type for string literals.

Default
The default with **xlC**, **xlc** and **c89** is **ro**. The default with **cc** is **noro**.

Notes
If **ro** is specified, the compiler places string literals in read-only storage. If **noro** is specified, string literals are placed in read/write storage.

You can also specify the storage type in your source program using:

```
#pragma strings storage_type
```

where *storage_type* is **read-only** or **writable**.

Placing string literals in read-only memory can improve runtime performance and save storage, but code that attempts to modify a read-only string literal generates a memory error.

Example
To compile myprogram.c so that the storage type is **writable**, enter:

```
xlC myprogram.c -qnoro
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

roconst

Option Type	Default Value	#pragma options	C	C++
-q*option*	roconst*	ROCONST	x	x

Syntax

```
-qroconst | -qnoroconst
ROCONST | NOROCONST
```

Purpose
Specifies the storage location for constant values.

Default
The default with **xlC**, **xlc** and **c89** is **roconst**. The default with **cc** is **noroconst**.

Notes
If **-qroconst** is specified, the compiler places constants in read-only storage. If **-qnoroconst** is specified, constant values are placed in read/write storage.

Placing constant values in read-only memory can improve runtime performance, save storage, and provide shared access. Code that attempts to modify a read-only constant value generates a memory error.

Constant value in the context of the -qroconst option refers to variables that are qualified by **const** (including const-qualified characters, integers, floats, enumerations, structures, unions, and arrays). The following variables do not apply to this option:
- variables qualified with **volatile** and aggregates (such as a **struct** or a **union**) that contain **volatile** variables
- pointers and complex aggregates containing pointer members
- automatic and static types with block scope
- uninitialized types
- regular structures with all members qualified by **const**
- initializers that are addresses, or initializers that are cast to non-address values

The **-qroconst** option does not imply the "ro" on page 210 option. Both options must be specified if you wish to specify storage characteristics of both string literals (**-qro**) and constant values (**-qroconst**).

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

rrm

Option Type	Default Value	#pragma options	C	C++
-q*option*	norrm	RRM	x	x

Syntax
```
-qrrm | -qnorrm
RRM | NORRM
```

Purpose
Prevents floating-point optimizations that are incompatible with run-time rounding to plus and minus infinity modes.

Notes
This option informs the compiler that, at run time, the floating-point rounding mode may change or that the mode is not set to "y" on page 242**n** (rounding to the nearest representable number.)

-qrrm must also be specified if the Floating Point Status and Control register is changed at run time.

The default, **-qnorrm**, generates code that is compatible with run-time rounding modes **nearest** and **zero**. For a list of rounding mode options, see the "y" on page 242 compiler option.

This option is obsolete. Use "float" on page 127**=rrm** in your new applications.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

rtti

Option Type	Default Value	#pragma options	C	C++
-q*option*	nortti	-		x

Syntax

```
-qrtti=option | -qnortti
```

Purpose
Use this option to generate run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.

Notes
For best run-time performance, suppresses RTTI information generation with the default **-qnorttis**etting.

The C++ language offers a (RTTI) mechanism for determining the class of an object at run time. It consists of two operators:

- one for determining the run-time type of an object (typeid), and,
- one for doing type conversions that are checked at run time (dynamic_cast).

A type_info class describes the RTTI available and defines the type returned by the typeid operator.

Suboptions are:

all	The compiler generates the information needed for the RTTI typeid and dynamic_cast operators. If you specify just -qrtti, this is the default suboption.
type | typeinfo	The compiler generates the information needed for the RTTI typeid operator, but the information needed for dynamic_cast operator is not generated.
dyna | dynamiccast	The compiler generates the information needed for the RTTI dynamic_cast operator, but the information needed for typeid operator is not generated.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

S

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-S
```

Purpose

Generates an assembler language file (**.s**) for each source file. The resulting **.s** files can be assembled to produce object **.o** files or an executable file (**a.out**).

Notes

You can invoke the assembler with the **xlC** command. For example,

```
xlC myprogram.s
```

will invoke the assembler, and if successful, the loader to create an executable file, **a.out**.

If you specify **-S** with "E" on page 116 or "P" on page 195, **-E** or **-P** takes precedence. Order of precedence holds regardless of the order in which they were specified on the command line.

You can use the "o" on page 190 option to specify the name of the file produced only if no more than one source file is supplied. For example, the following is *not* valid:

```
xlC myprogram1.c myprogram2.c -o -S
```

Restrictions

The generated assembler files do not include all the data that is included in a **.o** file by the "g" on page 135 or "ipa (C Only)" on page 151 options.

Example

To compile myprogram.c to produce an assembler language file **myprogram.s**, enter:

```
xlC myprogram.c -S
```

To assemble this program to produce an object file **myprogram.o**, enter:

```
xlC myprogram.s -c
```

To compile myprogram.c to produce an assembler language file **asmprogram.s**, enter:

```
xlC myprogram.c -S -o asmprogram.s
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

"Resolving Conflicting Compiler Options" on page 43
AIX Version 4 Assembler Language Reference
AIX Version 4 Files Reference

s

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-s
```

Purpose
This option strips the symbol table, line number information, and relocation information from the output file. Specifying -s saves space, but limits the usefulness of traditional debug programs when you are generating debug information using options such as -g.

Notes
Using the strip command has the same effect.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

showinc

Option Type	Default Value	#pragma options	C	C++
-q*option*	noshowinc	SHOwinc	x	x

Syntax

```
-qshowinc | -qnoshowinc
SHOwinc
```

Purpose
If used with "source" on page 217, all the include files are included in the source listing.

Example
To compile myprogram.c so that all included files appear in the source listing, enter:

```
xlC myprogram.c -qsource -qshowinc
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

smp (C Only)

Option Type	Default Value	#pragma options	C	C++
-q*option*	nosmp	-	x	

Syntax

```
-qnosmp | -qsmp[=suboption[:suboption] [ ... ]]
```

Purpose

Specifies if and how parallelized object code is generated, according to *suboption(s)* specified:

Suboption	Description
`auto` `noauto`	Enables or disables automatic parallelization. **auto** is the default if **-qsmp** is specified without the **omp** suboption. Otherwise, the default is **noauto**.
`explicit` `noexplicit`	Enables or disables pragmas controlling explicit parallelization of countable loops. **explicit** is the default. If **noexplicit** is in effect, **#pragma ibm omp parallel_loop** is not honored by the compiler.
`nested_par` `nonested_par`	Enables or disables parallelization of nested parallel constructs. **nonested_par** is the default. If one parallel construct is run as part of another parallel construct, the execution of the nested construct is serialized by the compiler for better performance. If **nested_par** is in effect, nested parallel constructs are not serialized. **Notes:** 1. **nested_par** does not provide true nested parallelism because it does not cause new team of threads to be created for nested parallel regions. Instead, threads that are currently available are re-used. 2. This option should be used with caution. Depending on the number of threads available and the amount of work in an outer loop, inner loops could be executed sequentially even if this option is in effect. Parallelization overhead may not necessarily be offset by program performance gains.

`omp` `noomp`	Enables or disables strict compliance with OpenMP C and C++ API specifications. **noomp** is the default. This mode allows for maximum program parallelization, but may not be completely compliant to the OpenMP API specification. If you specify the **omp** suboption, the compiler disables automatic parallelization and warns of directives that are not OpenMP-compliant. The _OPENMP macro is defined. Certain other **smp** suboptions enable compiler parallelization features that do not comply with the OpenMP specification. If they are specified together with the **omp** suboption, a warning message issued. These suboptions are: • auto • nested_par • rec_locks • schedule=affinity=*n*
`rec_locks` `norec_locks`	Specifies whether recursive locks are used to implement critical sections. If **rec_locks** is in effect, recursive locks are used, and nested critical sections will not cause a deadlock. The default is **norec_locks**, or regular locks.
`schedule=`*`sched_type`*`[=`*`n`*`]`	Specifies what kind of scheduling algorithms and chunking are used for loops to which no other scheduling algorithm has been explicitly assigned in the source code. Valid options for *sched_type* are: • dynamic[=*n*] • guided[=*n*] • static[=*n*] • affinity[=*n*] • runtime If *sched_type* is not specified, **runtime** is assumed as the default setting. For more information about these scheduling algorithms, see schedule pragma.

Notes

- Specifying **-qsmp** without suboptions is equivalent to specifying **-qsmp=auto:explicit:noomp:norec_locks:nonested_par:schedule=runtime**.
- The **-qnosmp** default option setting specifies that no code should be generated for parallelization directives, though syntax checking will still be performed. Use **-qignprag=omp:ibm** to completely ignore parallelization directives.
- Specifying **-qsmp** defines the _IBMSMP preprocessing macro

- Specifying **-qsmp** implicitly sets **-O2**. The **-qsmp** option overrides **-qnooptimize**, but does not override **-O3** or **-O4**.
- **-qsmp** must be used only with thread-safe compiler mode invocations such as **xlc_r**. These invocations ensure that the **pthreads**, **xlsmp**, and thread-safe versions of all default run-time libraries are linked to the resulting executable.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"IBM Run-time Options for Parallel Processing (C Only)" on page 63
"Built-in Functions Used for Parallel Processing (C Only)" on page 61
"O, optimize" on page 192 Optimize Option
"threaded" on page 229 Option
Equivalent Batch Compile-Link and Incremental Build Options

source

Option Type	Default Value	#pragma options	C	C++
-q*option*	nosource	SOURCE	x	x

Syntax:

```
-qsource | -qnosource
SOURCE | NOSOURCE
```

Purpose
Produces a compiler listing and includes source code.

Notes
The "noprint" on page 189 option overrides this option.

Parts of the source can be selectively printed by using pairs of **#pragma options source** and **#pragma options nosource** preprocessor directives throughout your source program. The source following **#pragma options source** and preceding **#pragma options nosource** is printed.

Example
The following code causes the parts of the source code between the **#pragma options** directives to be included in the compiler listing:

```
#pragma options source
   . . .
/* Source code to be included in the compiler listing
   is bracketed by #pragma options directives.
*/
   . . .
#pragma options nosource
```

To compile myprogram.c to produce a compiler listing that includes the source for **myprogram.c**, enter:

```
xlC myprogram.c -qsource
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

spill

Option Type	Default Value	#pragma options	C	C++
-q*option*	spill=512	SPILL=*size*	x	x

Syntax

```
-qspill=size
SPILL=size
```

Purpose
Specifies the register allocation spill area as being *size* bytes.

Notes
If your program is very complex, or if there are too many computations to hold in registers at one time and your program needs temporary storage, you might need to increase this area. Do not enlarge the spill area unless the compiler issues a message requesting a larger spill area. In case of a conflict, the largest spill area specified is used.

Example
If you received a warning message when compiling myprogram.c and want to compile it specifying a spill area of **900** entries, enter:

```
xlC myprogram.c -qspill=900
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

spnans

Option Type	Default Value	#pragma options	C	C++
-q*option*	nospnans	SPNANS	x	x

Syntax

```
-qspnans | -qnospnans
SPNANS | NOSPNANS
```

Purpose
Generates extra instructions to detect signalling NaN on conversion from single precision to double precision. The **nospnans** option specifies that this conversion need not be detected.

Notes
The "hsflt" on page 140 option overrides the **spnans** option

This option is obsolete. Use “float” on page 127**=nans** in your new applications.

RELATED REFERENCES

“List of Batch Compiler Options and Their Defaults” on page 24
Equivalent Batch Compile-Link and Incremental Build Options

srcmsg (C Only)

Option Type	Default Value	#pragma options	C	C++
-q*option*	nosrcmsg	SRCMSG	x	

Syntax

```
-qsrcmsg | -qnosrcmsg
SRCMSG | NOSRCMSG
```

Purpose
Adds the corresponding source code lines to the diagnostic messages in the **stderr** file.

Notes
The compiler reconstructs the source line or partial source line to which the diagnostic message refers and displays it before the diagnostic message. A pointer to the column position of the error may also be displayed. Specifying **-qnosrcmsg** suppresses the generation of both the source line and the finger line, and the error message simply shows the file, line and column where the error occurred.

The reconstructed source line represents the line as it appears after macro expansion. At times, the line may be only partially reconstructed. The characters “....” at the start or end of the displayed line indicate that some of the source line has not been displayed.

The default (**nosrcmsg**) displays concise messages that can be parsed. Instead of giving the source line and pointers for each error, a single line is displayed, showing the name of the source file with the error, the line and character column position of the error, and the message itself.

Example
To compile myprogram.c so that the source line is displayed along with the diagnostic message when an error occurs, enter:

```
xlC myprogram.c -qsrcmsg
```

RELATED REFERENCES

“List of Batch Compiler Options and Their Defaults” on page 24
“Options that Specify Compiler Output” on page 40
Equivalent Batch Compile-Link and Incremental Build Options

staticinline

Option Type	Default Value	#pragma options	C	C++
-q*option*	-qnostaticinline	-		x

Syntax

```
-qstaticinline | -qnostaticinline
```

This option controls whether inline functions are treated as static or extern.

By default, VisualAge C++; treats inline functions as extern.

For example, using the -qstaticinline option causes function f in the following declaration to be treated as static, even though it is not explicitly declared as such.

```
inline void f() {/*...*/};
```

Using the default, -qnostaticinline, gives f external linkage.

RELATED REFERENCES

Equivalent Batch Compile-Link and Incremental Build Options

statsym

Option Type	Default Value	#pragma options	C	C++
-q*option*	nostatsym	-	x	x

Syntax

```
-qstatsym | -qnostatsym
```

Purpose
Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the name list (the symbol table of **xcoff** objects).

Default
The default is to not add static variables to the symbol table. However, static functions are added to the symbol table.

Example
To compile myprogram.c so that static symbols are added to the symbol table, enter:

```
xlC myprogram.c -qstatsym
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

stdinc

Option Type	Default Value	#pragma options	C	C++
-q*option*	stdinc	STDINC	x	x

Syntax:

```
-qstdinc | -qnostdinc
STDINC | NOSTDINC
```

Purpose
Specifies which directories are used for files included by the **#include <***file_name***>** and **#include "***file_name***"** directives. The -qnostdinc option excludes the standard include directies (**/usr/include** for C and **/usr/vacpp/includ**e, **/usr/include** for C++) from the search.

Notes
If you specify **-qnostdinc**, the compiler will not search the directory **/usr/include** for C files, or the directories **/usr/vacpp/include** and **/usr/include** for C++ files, unless you explicitly add them with the "I" on page 141*directory* option.

If a full (absolute) path name is specified, this option has no effect on that path name. It will still have an effect on all relative path names.

-qnostdinc is independent of "idirfirst" on page 142. (**-qidirfirst** searches the directory specified with "I" on page 141*directory* before searching the directory where the current source file resides.

The search order for files is described in Directory Search Sequence for Include Files Using Relative Path Names.

The last valid **#pragma options [NO]STDINC** remains in effect until replaced by a subsequent **#pragma options [NO]STDINC**.

Example
To compile myprogram.c so that the directory **/tmp/myfiles** is searched for a file included in myprogram.c with the **#include "myinc.h"** directive, enter:

```
    xlC myprogram.c -qnostdinc -I/tmp/myfiles
```

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

strict

Option Type	Default Value	#pragma options	C	C++
-q*option*	see below	STRICT	x	x

Syntax

```
-qstrict | -qnostrict
STRICT | NOSTRICT
```

Purpose
Turns off the aggressive optimizations that have the potential to alter the semantics of your program. The default is -qnostrict for **"O, optimize" on page 192** and higher optimizations. It is -qstrict for "O, optimize" on page 192.

Notes
-qstrict turns off the following optimizations:
- Performing code motion and scheduling on computations such as loads and floating-point computations that may trigger an exception.
- Relaxing conformance to IEEE rules.
- Reassociating floating-point expressions.

This option is only valid with "O, optimize" on page 192 or higher optimization levels.

-qstrict sets **"float" on page 127=nofltint:nosqrt**.

-qnostrict sets **-qfloat=fltint:sqrt**.

You can use **-qfloat=fltint** and **-qfloat=rsqrt** to override the **-qstrict** settings.

For example:

- Using **"O, optimize" on page 192 -qstrict "float" on page 127=fltint** means that **-qfloat=fltint** is in effect, but there are no other aggressive optimizations.
- Using **-O3 -qnostrict -qfloat=norsqrt** means that the compiler performs all aggressive optimizations except **-qfloat=rsqrt**.

If there is a conflict between the options set with **-qnostrict** and **"float" on page 127=options**, the last option specified is recognized.

Example
To compile myprogram.c so that the aggressive optimizations of "O, optimize" on page 192 are turned off, range checking is turned off (**"float" on page 127=fltint**), and division by the result of a square root is replaced by multiplying by the reciprocal (**-qfloat=rsqrt**), enter:

```
xlC myprogram.c -O3 -qstrict -qfloat=fltint:rsqrt
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

strict_induction

Option Type	Default Value	#pragma options	C	C++
-q*option*	*See below.*	-	x	x

Syntax

```
-qstrict_induction | -qnostrict_induction
```

Purpose
Disables loop induction variable optimizations that have the potential to alter the semantics of your program. Such optimizations can change the result of a program if truncation or sign extension of a loop induction variable should occur as a result of variable overflow or wrap-around.

Notes
Use of this option is generally not recommended because it can cause considerable performance degradation. If your program is not sensitive to induction variable overflow or wrap-around, you should consider using **-qnostrict_induction** in conjunction with the "O, optimize" on page 192 optimization option.

Default

- **-qnostrict_induction** with optimization levels 3 or higher.
- **-qstrict_induction** otherwise.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

suppress

Option Type	Default Value	#pragma options	C	C++
-q*option*	nosuppress		x	x

Syntax

```
-qsuppress=nnnn-mmm[:nnnn-mmm...] | -qnosuppress
-qsuppress=nnnn-mmm[:nnnn-mmm...] | -qnosuppress=nnnn-mmm[:nnnn-mmm...]
```

Purpose
Prevents the specified batch compiler or driver informational or warning messages from being displayed or added to the listings.

Notes
To suppress IPA messages, enter **-qsuppress**before **-qipa** on the command line.

This option suppresses compiler messages only, and has no effect on linker or operating system messages.

Compiler messages that cause compilation to stop, such as (S) and (U) level messages, or other messages depending on the setting of the **-qhalt** compiler option, cannot be suppressed. For example, if the **-qhalt=w** compiler option is set, warning messages will not be suppressed by the **-qsuppress** compiler option.

The **-qnosuppress** compiler option cancels previous settings of **-qsuppress**.

Example
If your program normally results in the following output:

"t.c", line 1.1:1506-224 (I) Incorrect #pragma ignored

You can suppress the message by compiling with:

```
xlC myprogram.c -qsuppress=1506-224
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

symtab

Option Type	Default Value	#pragma options	C	C++
-q*option*	-	-	x	x

Syntax

```
-qsymtab=options
```

Purpose

Controls symbol table.

The following table describes the effect of specifying -qsymtabt=*option* for each of the following options.

Option	Effect
unref	Specifies that all **typedef** declarations, **struct**, **union**, and **enum** type definitions are included for processing by the Distributed Debugger. Use this option with the "g" on page 135 option to produce additional debugging information for use with the Distributed Debugger. When you specify the "g" on page 135 option, debugging information is included in the object file. To minimize the size of object and executable files, the compiler only includes information for symbols that are referenced. Debugging information is not produced for unreferenced arrays, pointers, or file-scope variables unless **-qsymtab=unref** is specified. Using **-qsymtab=unref** may make your object and executable files larger.
static	Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the name list (the symbol table of **xcoff** objects). The default is to not add static variables to the symbol table.

Examples
To compile myprogram.C so that static symbols are added to the symbol table, enter:

```
xlC myprogram.C -qsymtab=static
```

To include all symbols in myprogram.C in the symbols table for use with the Distributed Debugger, enter:

```
xlC myprogram.C -g -qsymtab=unref
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
Equivalent Batch Compile-Link and Incremental Build Options

syntaxonly (C Only)

Option Type	Default Value	#pragma options	C	C++
-q*option*	-	-	x	

Syntax

```
-qSYNTAXonly
```

Purpose
Causes the compiler to perform syntax checking without generating an object file..

Notes
The "P" on page 195, **-**"E" on page 116, and "C" on page 103 options override the **-qsyntaxonly** option, which in turn overrides the "c" on page 103 and "o" on page 190 options.

The **-qsyntaxonly** option suppresses only the generation of an object file. All other files (listings, precompiled header files, etc) are still produced if their corresponding options are set.

Example
To check the syntax of myprogram.c without generating an object file, enter:

```
xlC myprogram.c -qsyntaxonly
```

or

```
xlC myprogram.c -o testing -qsyntaxonly
```

Note that in the second example, the **-qsyntaxonly** option overrides the "o" on page 190 option so no object file is produced.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
Equivalent Batch Compile-Link and Incremental Build Options

tabsize

Option Type	Default Value	#pragma options	C	C++
-q*option*	tabsize=8	-	x	x

Syntax

```
-qtabsize=n
```

Purpose
Changes the length of tabs as perceived by the compiler.

Notes
n is the number of character spaces representing a tab in your source program.

This option only affects error messages that specify the column number at which an error occurred. For example, the compiler will consider tabs as having a width of one character if you specify **-qtabsize=1**. In this case, you can consider one character position (where each character and each tab equals one position, regardless of tab length) as being equivalent to one character column.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
Equivalent Batch Compile-Link and Incremental Build Options

tbtable

Option Type	Default Value	#pragma options	C	C++
-q*option*	(see below)	TBTABLE	x	x

Syntax

```
-qtbtable=suboption
TBTABLE=suboption
```

Purpose
Generates a traceback table that contains information about each function, including the type of function as well as stack frame and register information. The traceback table is placed in the text segment at the end of its code.

Notes
`Values for suboption are:`

`none`	No traceback table is generated. The stack frame cannot be unwound so exception handling is disabled.
`full`	A full traceback table is generated, complete with name and parameter information. This is the default if -qnoopt or -g are specified.
`small`	The traceback table generated has no name or parameter information, but otherwise has full traceback capability. This is the default if you have specified optimization and have not specified -g.

The **#pragma** options directive must be specified before the first statement in the compilation unit.

Default
Many performance measurement tools require a full traceback table to properly analyze optimized code. The /etc/vac.cfg batch compiler configuration file contains entries to accomodate this requirement. If you do not require full traceback tables for your optimized code, you can save file space by making the following changes to your **/etc/vac.cfg** batch compiler configuration file:

1. Remove the **-qtbtable=full** option from the **options** lines of the C or C++ compilation stanzas.
2. Remove the **-qtbtable=full** option from the **xlCopt** line of the **DFLT** stanza.

With these changes, the defaults for the **tbtable** option are:

- When compiling with optization options set, **-qtbtable=small**

- When compiling with no otimization options set, **-qtbtable=full**

See Interlanguage Calls - Traceback Table for a brief description of traceback tables. The AIX Version 4 traceback mechanism is described in the "Subroutine Linkage Convention" section of the *AIX Version 4 Assembler Language Reference*.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Debugging Features" on page 39
Equivalent Batch Compile-Link and Incremental Build Options

t

Option Type	Default Value	#pragma options	C	C++
-flag	*See below.*	-	x	x

Syntax:

```
-tprograms
```

Purpose
Adds the prefix specified by the "B" on page 99 option to the designated *programs*.

Notes
This option can only be used with the "B" on page 99 option. The flags representing the standard *program* names are:

Program	Description
c	Compiler front end
b	Compiler back end
p	Compiler preprocessor
a	Assembler
I	Interprocedural Analysis tool - compile phase
L	Interprocedural Analysis tool - link phase
l	Linkage editor
E	CreateExportList utility
m	Linkage helper (**munch** utility)

Default
If "B" on page 99 is specified but *prefix* is not, the default prefix is **/lib/o**. If **-B***prefix* is not specified at all, the prefix of the standard program names is **/lib/n**.

If "B" on page 99 is specified but **-t***programs* is not, the default is to construct path names for all the standard program names: (**c**,**b**, **I**, **a**, **l**, and **m**).

Example
To compile myprogram.c so that the name **/u/newones/compilers/** is prefixed to the compiler and assembler program names, enter:

```
xlC myprogram.c -B/u/newones/compilers/ -tca
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

tempinc

Option Type	Default Value	#pragma options	C	C++
-qoption	*See below.*	-		x

Syntax

```
-qtempinc | -qtempinc=directory | -qnotempinc
```

Purpose
Generates separate include files for template functions and class declarations, and places these files in a directory which can be optionally specified.

Notes
When you specify **-qtempinc**, the compiler assigns a value of 1 to the **__TEMPINC__** macro. This assignment will not occur if **-qnotempinc** has been specified.

Default
The default is to generate the separate include files and place them in the **tempinc** directory of the current directory at compile time.

Example
To compile the file myprogram.C and place the generated include files for the template functions in the **/tmp/mytemplates** directory, enter:

```
xlC myprogram.C -qtempinc=/tmp/mytemplates
```

RELATED CONCEPTS
Template Instantiation with VisualAge C++
RELATED REFERENCES
"tempmax"
"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

tempmax

Option Type	Default Value	#pragma options	C	C++
-flag	tempmax=1	-		x

Syntax

```
-qtempmax=number
```

Purpose
Specifies the maximum number of template include files to be generated by the "tempinc" option for each header file.

Notes
Specify the maximum number of template files by giving *number* a value between 1 and 99999.

Instantiations are spread among the template include files.

This option should be used when the size of files generated by the "tempinc" on page 228 option become very large and take a significant amount of time to recompile when a new instance is created.

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24

threaded

Option Type	Default Value	#pragma options	C	C++
-q*option*	*See Below*	-	x	x

Syntax

```
-qthreaded | -qnothreaded
```

Purpose
Indicates to the compiler that the program will run in a multi-threaded environment. Always use this option when compiling or linking multi-threaded applications. This option ensures that all optimizations are thread-safe.

Notes
This option applies to both compile and linkage editor operations.

To maintain thread safety, a file compiled with the **-qthreaded** option, whether explicitly by option selection or implicitly by choice of **_r** compiler invocation mode, must also be linked with the **-qthreaded** option.

This option does not make code thread-safe, but it will ensure that code already thread-safe will remain so after compile and linking.

Default
The default is **-qthreaded** when compiling with **_r** invocation modes, and **-qnothreaded** when compiling with other invocation modes.

RELATED TASKS
"Specify Batch Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 21

RELATED REFERENCES
"Compiler Modes" on page 1
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
"smp (C Only)" on page 215
Equivalent Batch Compile-Link and Incremental Build Options

tune

Option Type	Default Value	#pragma options	C	C++
-q*option*	*See below.*	TUNE=*suboption*	x	x

Syntax

```
-qtune=suboption
TUNE=suboption
```

Purpose

Specifies the architecture system for which the executable program is optimized.

Notes

Allowable values for *suboption* are:

auto	Produces object code optimized for the hardware platfom on which it is compiled.
`403`	Produces object code optimized for the PowerPC 403 processor.
`601`	Produces object code optimized for the PowerPC 601® processor.
`603`	Produces object code optimized for the PowerPC 603™ processor.
`604`	Produces object code optimized for the PowerPC 604™ processor.
`p2sc`	Produces object code optimized for the PowerPC P2SC processor.
`pwr`	Produces object code optimized for the POWER hardware platforms.
`pwr2`	Produces object code optimized for the POWER2 hardware platforms.
`pwr2s`	Produces object code optimized for the POWER2 hardware platforms, avoiding certain quadruple-precision instructions that would slow program performance.
`pwr3`	Produces object code optimized for the POWER3 hardware platforms.
`pwrx`	Produces object code optimized for the POWER2 hardware platforms (same as **-qtune=pwr2**).
rs64a	Produces object code optimized for the RS64A processor.
rs64b	Produces object code optimized for the RS64B processor.
rs64c	Produces object code optimized for the RS64C processor.

If **-qtune** is specified without “arch” on page 96**=***suboption*, the compiler uses **-qarch=com**.

You can use **-qtune=***suboption* with “arch” on page 96**=***suboption*.

- **-qarch=***suboption* specifies the architecture for which the instructions are to be generated, and,
- **-qtune=***suboption* specifies the target platform for which the code is optimized.

Default
The default setting of the **-qtune=** option depends on the setting of the "arch" on page 96= option.

- If **-qtune** is specified without **-qarch**, the compiler uses **-qarch=com**.
- If **-qarch** is specified without **-qtune=**, the compiler uses the default tuning option for the specified architecture. Listings will show only:

```
TUNE=DEFAULT
```

To find the actual default **-qtune** setting for a given **-qarch** setting, refer to the Acceptable Compiler Mode and Processor Architecture Combinations table.

Example
To specify that the executable program testing compiled from myprogram.c is to be optimized for a POWER hardware platform, enter:

```
xlC -o testing myprogram.c -qtune=pwr
```

RELATED TASKS

"Specify Batch Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 21

RELATED REFERENCES

Acceptable Compiler Mode and Processor Architecture Combinations
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

twolink

Option Type	Default Value	#pragma options	C	C++
-flag	notwolink*	-		x

Syntax

```
-qtwolink | -qnotwolink
```

Purpose
Minimizes the number of static constructors included from libraries.

Notes
Normally, the compiler links in all static constructors defined anywhere in the object (.o) files and library (.a) files. The **-qtwolink** option makes link time take longer, but linking is compatible with older versions of C or C++ compilers.

Before using **-qtwolink**, make sure that any .o files placed in an archive do not change the behavior of the program.

Default
The default is **notwolink**. All static constructors in .o files and object files are invoked. This generates larger executable files, but ensures that placing a .o file in a library does not change the behavior of a program.

Example
Given the include file foo.h:

```
#include <stdio.h>
struct foo {
    foo() {printf ("in foo\n");}
    ~foo() {printf ("in ~foo\n");}
};
```

and the C++ program t.C:

```
#include "foo.h"
foo bar;
```

and the program t2.C:

```
#include "foo.h"
main() { }
```

Compile t.Cc and t2.C in two steps, first invoking the compiler to produce object files:

```
xlC -c t.C t2.C
```

and then link them to produce the executable file a.out:

```
xlC t.o t2.o
```

Invoking a.out produces:

```
in foo
in ~foo
```

If you use the AIX **ar** command with the t.o file to produce an archive file t.a:

```
ar rv t.a t.o
```

and then use the default compiler command:

```
xlC t2.o t.a
```

The output from the executable file is the same as above:

```
in foo
in ~foo
```

However, if you use the **-qtwolink** option:

```
xlC -qtwolink t2.o t.a
```

there is no output from the executable file a.out becuase the static constructor foo() in t.C is not found.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

U

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-Uname
```

Purpose
Undefines the identifier *name* defined by the compiler or by the "D" on page 112*name* option.

Notes
The **-U***name* option is *not* equivalent to the **#undef** preprocessor directive. It *cannot* undefine names defined in the source by the **#define** preprocessor directive. It can only undefine names defined by the compiler or by the "D" on page 112*name* option.

The identifier name can also be undefined in your source program using the **#undef** preprocessor directive.

The **-U***name* option has a higher precedence than the "D" on page 112*name* option.

Example
To compile myprogram.c so that the definition of the name **COUNT**, is nullified, enter:

```
xlC myprogram.c  -UCOUNT
```

For example if the option **-DCOUNT=1000** is used, a source line **#undefine COUNT** is generated at the top of the source.

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Preprocessor Options" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

unique

Option Type	Default Value	#pragma options	C	C++
-flag	nounique	-		x

Syntax

```
-qunique | -qnounique
```

Purpose
Generates unique names for static constructor/deconstructor file compilation units.

Notes
Unique names are generated with **-qunique** by encoding random numbers into thename of the static initialization (sinit) and static termination (sterm) functions. Default behavior is encoding the absolute path name of the source file in the sinit and sterm functions. If the absolute path name will be identical for multiple compilations (for example, if a **make** script is used), the **-qunique** option is necessary.

Using **-qnounique** allows you to do incremental linking of files.

If you use **-qunique**, you must always link with all **.o** and **.a** files. Do not include an executable file on the link step. An object (**.o**) file compiled with **-qunique** should not be used for incremental linking.

Example
Suppose you want to compile several files using the same path name, ensuring that static construction works correctly. A **make** file may generate the following steps:

```
sqlpreprocess file1.sql > t.C
  xlC -qunique t.C -o file1.o
rm -f t.C
sqlpreprocess file2.sql > t.C
  xlC -qunique t.C -o file2.o
rm -f t.C
  xlC file1.o file2.o
```

Following is a sample **make** file for the above example:

```
# rule to get from file.sql to file.o
.SUFFIXES:     .sql
.sql.o:
       sqlpreprocess $< > t.C
       $(CCC) t.C -c $(CCFLAGS) -o $@
       rm -f t.C
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

unroll

Option Type	Default Value	#pragma options	C	C++
-q*option*	see below	-	x	x

Syntax

```
-qunroll=n | -qnounroll
```

Purpose
Unrolls inner loops in the program by a factor of *n*. By default, the optimizer selects the best value for each loop.

Notes
When **-qunroll** is specified, the bodies of inner loops will be duplicated *n*-1 times, creating a loop with *n* original bodies. The loop control may be modified in some cases to avoid unnecessary branching.

The maximum value for *n* is 8.

Example
In the following example, loop control is not modified:

```
while (*s != 0)
{
  *p++ = *s++;
}
```

Unrolling this by a factor of 2 gives:

```
while (*s)
{
  *p++ = *s++;
  if (*s == 0) break;
  *p++ = *s++;
}
```

In this example, loop control *is* modified:

```
for (i=0; i<n; i++) {
  a[i]=b[i] * c[i];
}
```

Unrolling by 3 gives:

```
i=0;
if (i>n-2) goto remainder;
for (; i<n-2; i+=3) {
  a[i]=b[i] * c[i];
  a[i+1]=b[i+1] * c[i+1];
  a[i+2]=b[i+2] * c[i+2];
}
if (i<n) {
  remainder:
  for (; i<n; i++) {
    a[i]=b[i] * c[i];
  }
}
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
Equivalent Batch Compile-Link and Incremental Build Options

upconv

Option Type	Default Value	#pragma options	C	C++
-q*option*	noupconv*	UPCONV	x	

Syntax

```
-qupconv | -qnoupconv
UPCONV | NOUPCONV
```

Purpose

Preserves the **unsigned** specification when performing integral promotions.

Notes

The **-qupconv** option promotes any **unsigned** type smaller than an **int** to an **unsigned int** instead of to an **int**.

Unsignedness preservation is provided for compatibility with older dialects of C. The ANSI C standard requires value preservation as opposed to unsignedness preservation.

Default

The default is **-qnoupconv**, except when "langlvl" on page 160**=ext**, in which case the default is **-qupconv**. The compiler does not preserve the **unsigned** specification.

The default compiler action is for integral promotions to convert a **char**, **short int**, **int bitfield** or their **signed** or **unsigned** types, or an **enumeration** type to an **int**. Otherwise, the type is converted to an **unsigned int**.

Example
To compile myprogram.c so that all **unsigned** types smaller than an **int** are converted to **unsigned int**, enter:

```
xlC myprogram.c -qupconv
```

The following short listing demonstrates the effect of **-qupconv**:

```
#include <stdio.h>
int main(void) {
  unsigned char zero = 0;
  if (-1 <zero)
    printf("Value-preserving rules in effect\n");
  else
    printf("Unsignedness-preserving rules in effect\n");
  return 0;
}
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

usepcomp

Option Type	Default Value	#pragma options	C	C++
-q*option*	nousepcomp	-	x	

Syntax

```
-qusepcomp | -qusepcomp=directory | -qnousepcomp
```

Purpose
Uses a precompiled header file if no included files that have not changed since the precompiled header was created. This may help improve compile time.

Notes
Usage modes for **usepcomp** are:

-qusepcomp	Uses the precompiled header file called **csetc.pch**, if it exists in the current directory.
-qusepcomp=*directory*	Uses a precompiled header file if: • *directory* is the name of an existing directory, and the **csetc.pch** precompiled header file exists in that directory. • a directory with the name *directory* does not exist, but a precompiled header file called *directory* exists in the current directory.
-qnousepcomp	Does not use precompiled header files.

The **-qusepcomp** and "genpcomp" on page 136 options are designed to be used together, but they may be used separately.

- **-qgenpcomp** used alone will refresh the contents of the precompiled header file, even if it already exists. This is useful if the file has been corrupted.
- **-qusepcomp** used alone will use an existing precompiled header file without creating a new one. This is useful if you only want do not want the precompiled header file to be recompiled, or if remaining disk space is low.

When **-qusepcomp** and "genpcomp" on page 136 are used together, the compiler will automatically maintain and use a current precompiled header.

If you update your system header files, you can regenerate them with the **/usr/vacpp/bin/mkpcomp** command.

Precompiled headers will only be used at the same language level used during their creation.

For a given #include, **-qusepcomp** is checked first. Then the compiler checks for a precompiled version of the file to be included if such is specified. If it is found and it is current, it is used.

If a precompiled header is not being used (for example, if a current one is not found, or if **-qusepcomp** is not specified), and "genpcomp" on page 136 is specified, the compiler will create a new precompiled header (even if it exists and is current).

The precompiled headers created by installing IBM VisualAge C++ are listed in the LPP inventory, and are removed if you uninstall IBM VisualAge C++. Any additional headers you create are *not* removed during uninstall.

RELATED CONCEPTS

Precompiled C Headers

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

V

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-V
```

Purpose
Instructs the compiler to report information on the progress of the compilation, names the programs being invoked within the compiler and the options being specified to each program. Information is displayed in a format similar to that of shell commands.

Notes
The **-V** option is overridden by the "#" on page 89 option.

Example
To compile myprogram.C so you can watch the progress of the compilation and see messages that describe the progress of the compilation, the programs being invoked, and the options being specified, enter:

```
xlC myprogram.C -V
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

v

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-v
```

Purpose
Instructs the compiler to report information on the progress of the compilation, names the programs being invoked within the compiler and the options being specified to each program. Information is displayed to standard output.

Notes
The **-v** option is overridden by the "#" on page 89 option.

Example
To compile myprogram.c so you can watch the progress of the compilation and see messages that describe the progress of the compilation, the programs being invoked, and the options being specified, enter:

```
xlC myprogram.c -v
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

vftable

Option Type	Default Value	#pragma options	C	C++
-q*opt*	*See below.*	-		x

Syntax

```
-qvftable | -qnovftable
```

Purpose
Controls the generation of virtual function tables.

Notes
Specifying **-qvftable** generates virtual function tables for all classes with virtual functions that are defined in the current compilation unit.

If you specify **-qnovftable**, no virtual function tables are generated in the current compilation unit.

Default
The default is to define the virtual function table for a class if the current compilation unit contains the body of the first non-inline virtual member function declared in the class member list.

Example
To compile the file myprogram.C so that no virtual function tables are generated, enter:

```
xlC myprogram.C -qnovftable
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

W

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-Wprogram, options
```

Purpose
Passes the listed options to the designated compiler *program*.

program can be:

program	Description
a	Assembler
b	Compiler back end
c	Compiler front end
I	Interprocedural Analysis tool
l	linkage editor
p	compiler preprocessor

Notes
When used in the configuration file, the **-W** option accepts the escape sequence backslash comma (**\,**) to represent a comma in the parameter string.

Example
To compile myprogram.c so that the *option* **-pg** is passed to the linkage editor (**l**) and the assembler (**a**), enter:

```
xlC myprogram.c -Wl:a, -pg
```

In a configuration file, use the **\,** sequence to represent the comma (,).

```
-Wl:a\,-pg
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Characteristics" on page 38
Equivalent Batch Compile-Link and Incremental Build Options

w

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

```
-w
```

Purpose
Requests that warnings and lower-level messages be suppressed. Specifying this option is equivalent to specifying "flag" on page 126**=e:e**.

Example
To compile myprogram.c so that no warning messages are displayed, enter:

```
xlC myprogram.c -w
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

warn64

Option Type	Default Value	#pragma options	C	C++
-q*option*	nowarn64	-	x	x

Syntax

```
-qwarn64
```

Purpose
Enables checking for possible *long-to-integer* truncation.

Notes
All generated messages have level Informational.

This option functions in either 32- or 64-bit compiler modes. In 32-bit mode, it functions as a preview aid to discover possible 32- to 64-bit migration problems.

Informational messages are displayed where data conversion may cause problems in 64-bit compilation mode, such as:

- truncation due to explicit or implicit conversion of **long** types into **int** types
- unexpected results due to explicit or implicit conversion of **int** types into **long** types
- invalid memory references due to explicit conversion by cast operations of **pointer** types into **into** types
- invalid memory references due to explicit conversion by cast operations of **int** types into **pointer** types
- problems due to explicit or implicit conversion of **constants** into **long** types
- problems due to explicit or implicit conversion by cast operations of **constants** into **pointer** types
- conflicts with pragma options **arch** in source files and on the command line

RELATED TASKS

"Specify Batch Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 21

RELATED REFERENCES

Acceptable Compiler Mode and Processor Architecture Combinations
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41
"longlit" on page 176
Equivalent Batch Compile-Link and Incremental Build Options

xcall

Option Type	Default Value	#pragma options	C	C++
-q*option*	noxcall	-	x	x

Syntax

```
-qxcall | -qnoxcall
```

Purpose
Generates code to static routines within a compilation unit as if they were external routines.

Notes
-qxcall generates slower code than **-qnoxcall**.

Example
To compile myprogram.c so all static routines are compiled as external routines, enter:

```
xlC myprogram.c -qxcall
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

xref

Option Type	Default Value	#pragma options	C	C++

-q*option*	noxref	XREF	x	x

Syntax

```
    -qxref | -qnoxref
    XREF | NOXREF
```

Purpose
Produces a compiler listing that includes a cross-reference listing of all identifiers.

Notes
Usage modes for **xref** are:

`-qxref=full`	Reports all identifiers in the program.
`-qxref`	Reports only those identifiers that are used.

The "noprint" on page 189 option overrides this option.

Any function defined with the **#pragma mc_func***function_name* directive is listed as being defined on the line of the **#pragma** directive.

Example
To compile myprogram.c and produce a cross-reference listing of all identifiers whether they are used or not, enter:

```
     xlC myprogram.c -qxref=full -qattr
```

A typical cross-reference listing has the form:

RELATED REFERENCES
"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify Compiler Output" on page 40
Equivalent Batch Compile-Link and Incremental Build Options

y

Option Type	Default Value	#pragma options	C	C++
-flag	-	Y*rounding_mode*	x	x

Syntax

```
    -yrounding_mode
Yrounding_mode
```

Purpose
Specifies the compile-time rounding mode of constant floating-point expressions.

Notes
rounding_mode must be one of the following:

`n`	Round to the nearest representable number. This is the default.
`m`	Round toward minus infinity.

p	Round toward plus infinity.
z	Round toward zero.

Example
To compile myprogram.c so that constant floating-point expressions are rounded toward zero at compile time, enter:

```
    xlC myprogram.c -yz
```

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
"Options that Specify the Compiler Object Code Produced" on page 41 Equivalent Batch Compile-Link and Incremental Build Options

Z

Option Type	Default Value	#pragma options	C	C++
-flag	-	-	x	x

Syntax

`-Zstring`

Purpose
This option specifies a prefix for the library search path.

This option is useful when developing a new version of a library. Usually you use it to build on one level of AIX and run on a different level, so that you can search a different path on the development platform than on the target platform. This is possible because the prefix is not stored in the executable.

If use this option more than once, the strings are appended to each other in the order specified and then they are added to the beginning of the library search paths.

RELATED REFERENCES

"List of Batch Compiler Options and Their Defaults" on page 24
Equivalent Batch Compile-Link and Incremental Build Options

Appendix B. Pragmas for Parallel Processing

#pragma ibm critical Preprocessor Directive (C Only)

The *critical* pragma identifies a critical section of program code that must only be run by one process at a time.

Syntax

```
#pragma ibm critical [(name)]
<statement>
```

where *name* can be used to optionally identify the critical region. Identifiers naming a critical region have external linkage.

Notes
The compiler reports an error if you try to branch into or out of a critical section. Some situations that will cause an error are:

- A critical section that contains the **return** statement.
- A critical section that contains **goto**, **continue**, or **break** statements that transfer program flow outside of the critical section.
- A **goto** statement outside a critical section that transfers program flow to a label defined within a critical section.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"smp (C Only)" on page 215 Compiler Option

#pragma ibm independent_calls Preprocessor Directive (C Only)

The *independent_calls* pragma asserts that specified function calls within the chosen loop have no loop-carried dependencies. This information helps the compiler perform dependency analysis.

Syntax

```
#pragma ibm independent_calls [(identifier [,identifier] ... )]
<countable for/while/do loop>
```

where *identifier* represents the name of a function.

Notes
identifier cannot be the name of a pointer to a function.

If no function identifiers are specified, the compiler assumes that all functions inside the loop are free of carried dependencies.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60

#pragma ibm independent_loop Preprocessor Directive (C Only)

The *independent_loop* pragma asserts that iterations of the chosen loop are independent, and that the loop can be parallelized.

Syntax

```
#pragma ibm independent_loop [if (exp)]
<countable for/while/do loop>
```

where *exp* represents a scalar expression. When the **if** argument is specified, loop iterations are considered independent only as long as *exp* evaluates to TRUE at run-time.

Notes
This pragma can be combined with the **schedule** pragma to select a specific parallel process scheduling algorithm. For more information, see the description for the "#pragma ibm schedule Preprocessor Directive (C Only)" on page 248 pragma.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60

#pragma ibm iterations Preprocessor Directive (C Only)

The *iterations* pragma specifies the approximate number of loop iterations for the chosen loop.

Syntax

```
#pragma ibm iterations (iteration-count)
<countable for/while/do loop>
```

where *iteration-count* represents a positive integral constant expression.

Notes
The compiler uses the information in the *iteration-count* variable to determine if it is efficient to parallelize the loop.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60

#pragma ibm parallel_loop Preprocessor Directive (C Only)

The *parallel_loop* pragma explicitly instructs the compiler to parallelize the chosen loop.

Syntax

```
#pragma ibm parallel_loop [if (exp)] [schedule (sched-type)]
<countable for/while/do loop>
```

where *exp* represents a scalar expression, and *sched-type* represents any scheduling algorithm as valid for the *schedule* directive. When the **if** argument is specified, the loop executes in parallel only if *exp* evaluates to TRUE at run-time. Otherwise the loop executes sequentially. The loop will also run sequentially if it is in a critical section.

Notes
This pragma can be applied to a wide variety of C loops, and the compiler will try to determine if a loop is countable or not.

Program sections using the **parallel_loop** pragma must be able to produce a correct result in both sequential and parallel mode. For example, loop iterations must be independent before the loop can be parallelized. Explicit parallel programming techniques involving condition synchronization are not permitted.

This pragma can be combined with the **schedule** pragma to select a specific parallel process scheduling algorithm. For more information, see the description for the "#pragma ibm schedule Preprocessor Directive (C Only)" on page 248 pragma.

A warning is generated if this pragma is not followed by a countable loop.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

“Control Parallel Processing with Pragmas (C Only)” on page 59

RELATED REFERENCES

“#pragma Preprocessor Directives for Parallel Processing (C Only)” on page 60
“smp (C Only)” on page 215 Compiler Option

#pragma ibm permutation Preprocessor Directive (C Only)

The *permutation* pragma asserts that specified arrays in the chosen loop contain no repeated values.

Syntax

```
#pragma ibm permutation (identifier [,identifier] ... )
<countable for/while/do loop>
```

where *identifier* represents the name of an array.

Notes
identifier cannot be the name of a pointer.

An array specified by this pragma cannot be a function parameter.

RELATED CONCEPTS

“Chapter 2. Program Parallelization (C Only)” on page 53
“Shared and Private Variables in a Parallel Environment (C Only)” on page 57
“Countable Loops (C Only)” on page 54

RELATED TASKS

“Control Parallel Processing with Pragmas (C Only)” on page 59

RELATED REFERENCES

“#pragma Preprocessor Directives for Parallel Processing (C Only)” on page 60

#pragma ibm schedule Preprocessor Directive (C Only)

The *schedule* pragma specifies the scheduling algorithms used for parallel processing.

Syntax

```
#pragma ibm schedule (sched-type)
<countable for/while/do loop>
```

where *sched-type* represents one of the following options:

`affinity`
`affinity,n`
	Iterations of a loop are initially divided into local partitions of size **ceiling**(*number_of_iterations*/*number_of_threads*). Each local partition then further subdivided into chunks of size **ceiling**(*number_of_iterations_remaining_in_partition*/2). If *n* is specified, each local partition is subdivided into chunks of size *n*. *n* must be an integral assignment expression of value 1 or greater. When a thread becomes available, it takes the next chunk from its local partition. If there are no more chunks in the local partition, the thread takes an available chunk from the partition of another thread.

`dynamic`
`dynamic,n`
	If *n* is not specified, iterations of a loop are divided into chunks of size 1. If *n* is specified, all chunks are set to size *n*. *n* must be an integral assignment expression of value 1 or greater. Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

`guided`
`guided,n`
	Chunks are made progressively smaller until the default minimum chunk size is reached. The first chunk is of size **ceiling**(*number_of_iterations*/*number_of_threads*). Remaining chunks are of size **ceiling**(*number_of_iterations_remaining*/*number_of_threads*). If *n* is specified, the minimum chunk size is set to *n*. *n* must be an integral assignment expression of value 1 or greater. If *n* is not specified, a default value of 1 is assumed. Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

`runtime`	Scheduling policy is determined at run-time.
`static`	Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*). Each thread is assigned a separate chunk. This scheduling policy is also known as *block scheduling*.
`static,`*`n`*	Iterations of a loop are divided into chunks of size *n*. Each chunk is assigned to a thread in *round-robin* fashion. *n* must be an integral assignment expression of value 1 or greater. This scheduling policy is also known as *block cyclic scheduling*.
`static,1`	Iterations of a loop are divided into chunks of size 1. Each chunk is assigned to a thread in *round-robin* fashion. This scheduling policy is also known as *cyclic scheduling*.

Notes

Scheduling algorithms for parallel processing can be specified using any of the methods shown below. If used, methods higher in the list override entries lower in the list.

- pragma statements
- compiler command line options
- run-time command line options
- run-time default options

Scheduling algorithms can also be specified using the **schedule** argument of the "#pragma ibm parallel_loop Preprocessor Directive (C Only)" on page 247 pragma statements. For example, the following sets of statements are equivalent:

```
#pragma ibm parallel_loop
#pragma ibm schedule (sched_type)
<countable for|while|do loop>
```

and

```
#pragma ibm parallel_loop (sched_type)
<countable for|while|do loop>
```

If different scheduling types are specified for a given loop, the last one specified is applied.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"Built-in Functions Used for Parallel Processing (C Only)" on page 61
"#pragma ibm parallel_loop Preprocessor Directive (C Only)" on page 247
"IBM Run-time Options for Parallel Processing (C Only)" on page 63
"smp (C Only)" on page 215 Compiler Option

#pragma ibm sequential_loop Preprocessor Directive (C Only)

The *sequential_loop* pragma explicitly instructs the compiler to execute the chosen loop sequentially.

Syntax

```
#pragma ibm sequential_loop
<countable for/while/do loop>
```

Notes

This pragma disables automatic parallelization of the chosen loop, and is always respected by the compiler.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60

#pragma omp atomic Preprocessor Directive (C Only)

The **omp atomic** directive identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.

Syntax

```
    #pragma omp atomic
        statement
```

where *statement* is an expression statement of scalar type that takes one of the forms that follow:

x bin_op = expr	where:	
	bin_op	is one of: `+ * - / & ^	<< >>`
	expr	is an expression of scalar type that does not reference *x*.
x++		
++x		
x—		
—x		

Notes

Load and store operations are atomic only for object *x*. Evaluation of *expr* is not atomic.

All atomic references to a given object in your program must have a compatible type.

Objects that can be updated in parallel and may be subject to race conditions should be protected with the **omp atomic** directive.

Examples

```
extern float x[], *p = x, y;

/* Protect against race conditions among multiple updates.  */
#pragma omp atomic
x[index[i]] += y;

/* Protect against races with updates through x.            */
#pragma omp atomic
p[i] -= 1.0f;
```

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66

#pragma omp barrier Preprocessor Directive (C Only)

The **omp barrier** directive identifies a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point. Statement execution past the **omp barrier** point then continues in parallel.

Syntax

```
#pragma omp barrier
```

Notes
The **omp barrier** directive must appear within a block or compound statement. For example:

```
if (x!=0) {
   #pragma omp barrier    /* valid usage    */
}
if (x!=0)
   #pragma omp barrier    /* invalid usage  */
```

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66

#pragma omp critical Preprocessor Directive (C only)

The **omp critical** directive identifies a section of code that must be executed by a single thread at a time.

Syntax

```
#pragma omp critical [(name)]
   statement_block
```

where *name* can optionally be used to identify the critical region. Identifiers naming a critical region have external linkage and occupy a namespace distinct from that used by ordinary identifiers.

Notes
A thread waits at the start of a critical region identified by a given name until no other thread in the program is executing a critical region with that same name. Critical sections not specifically named by **omp critical** directive invocation are mapped to the same unspecified name.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
Countable Loops

RELATED TASKS
Control Parallel Processing with Pragmas

RELATED REFERENCES
"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66

#pragma omp flush Preprocessor Directive (C Only)

The **omp flush** directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.

Syntax

```
#pragma omp flush [ (list) ]
```

where *list* is a comma-separated list of variables that will be synchronized.

Notes
If *list* includes a pointer, the pointer is flushed, not the object being referred to by the pointer. If *list* is not specified, all shared objects are synchronized except those inaccessible with automatic storage duration.

An implied **flush** directive appears in conjuction with the following directives:
- **omp barrier**
- Entry to and exit from **omp critical**.
- Exit from **omp parallel**.
- Exit from **omp for**.
- Exit from **omp sections**.
- Exit from **omp single**.

The **omp flush** directive must appear within a block or compound statement. For example:

```
if (x!=0) {
   #pragma omp flush    /* valid usage    */
}
if (x!=0)
   #pragma omp flush    /* invalid usage  */
```

RELATED CONCEPTS
"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS
"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES
"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66

#pragma omp for Preprocessor Directive (C Only)

The **omp for** directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct.

Syntax

```
#pragma omp for [clause[ clause] ...]
<for_loop>
```

where *clause* is any of the following:

private (*list*)
: Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list*)
: Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

lastprivate (*list*)
: Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last iteration. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.

reduction (*operator*: *list*)
: Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

Variables specified in the **reduction** clause:

- must be of a type appropriate to the operator.
- must be shared in the enclosing context.
- must not be const-qualified.
- must not have pointer type.

ordered
: Specify this clause if an ordered construct is present within the dynamic extent of the **omp for** directive.

schedule (*type*) Specifies how iterations of the **for** loop are divided among available threads. Acceptable values for *type* are:

`dynamic`
`dynamic,n`
	If *n* is not specified, iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*).
	
	If *n* is specified, all chunks are set to size *n*. *n* must be an integral assignment expression of value 1 or greater.
	
	Chunks are dynamically assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

`guided`
`guided,n`
	Chunks are made progressively smaller until the default minimum chunk size is reached. The first chunk is of size **ceiling**(*number_of_iterations*/*number_of_threads*). Remaining chunks are of size **ceiling**(*number_of_iterations_remaining*/*number_of_threads*)
	
	If *n* is specified, the minimum chunk size is set to *n*. *n* must be an integral assignment expression of value 1 or greater.
	
	If *n* is not specified, a default value of 1 is assumed.
	
	Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

`runtime`	Scheduling policy is determined at run-time. Use the OMP_SCHEDULE environment variable to set the scheduling type and chunk size.
`static`	Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*). Each thread is assigned a separate chunk. This scheduling policy is also known as *block scheduling*.
`static,`*`n`*	Iterations of a loop are divided into chunks of size *n*. Each chunk is assigned to a thread in *round-robin* fashion. *n* must be an integral assignment expression of value 1 or greater. This scheduling policy is also known as *block cyclic scheduling*.
`static,1`	Iterations of a loop are divided into chunks of size 1. Each chunk is assigned to a thread in *round-robin* fashion. This scheduling policy is also known as *cyclic scheduling*.

nowait
Use this clause to avoid the implied **barrier** at the end of the **for** directive. This is useful if you have multiple independent work-sharing sections or iterative loops within a given parallel region. Only one **nowait** clause can appear on a given **for** directive.

and where *for_loop* is a **for** loop construct with the following canonical shape:

```
for (init_expr; exit_cond; incr_expr)
 statement
```

where:

	takes form:	
init_expr		*iv = b* integer-type iv = b
	takes form:	
exit_cond		*iv <= ub* *iv < ub* *iv >= ub* *iv > ub*
	takes form:	

incr_expr		++*iv*
		iv++
		–iv
		iv–
		iv += incr
		iv -= incr
		iv = iv + incr
		iv = incr + iv
		iv = iv - incr

iv	Iteration variable. The iteration variable must be a signed integer not modified anywhere within the **for** loop. It is implicitly made private for the duration of the **for** operation. If not specified as **lastprivate**, the iteration variable will have an indeterminate value after the operation completes..
b, *ub*, *incr*	Loop invariant signed integer expressions. No synchronization is performed when evaluating these expressions and evaluated side effects may result in indeterminate values..

Notes

Program sections using the **omp for** pragma must be able to produce a correct result regardless of which thread executes a particular iteration. Similarly, program correctness must not rely on using a particular scheduling algorithm.

The **for** loop iteration variable is implicitly made private in scope for the duration of loop execution. This variable must not be modified within the body of the **for** loop. The value of the increment variable is indeterminate unless the variable is specified as having a data scope of **lastprivate**.

An implicit barrier exists at the end of the **for** loop unless the **nowait** clause is specified.

Restrictions are:

- The **for** loop must be a structured block, and must not be terminated by a **break** statement.
- Values of the loop control expressions must be the same for all iterations of the loop.
- An **omp for** directive can accept only one **schedule** clauses.
- The value of *n* (chunk size) must be the same for all threads of a parallel region.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"#pragma omp ordered Preprocessor Directive (C Only)"
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66

#pragma omp master Preprocessor Directive (C Only)

The **omp master** directive identifies a section of code that must be run only by the master thread.

Syntax

```
#pragma omp master
   statement_block
```

Notes
Threads other than the master thread will not execute the statement block associated with this construct.

No implied barrier exists on either entry to or exit from the master section.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66

#pragma omp ordered Preprocessor Directive (C Only)

The **omp ordered** directive identifies a structured block of code that must be executed in sequential order.

Syntax

```
#pragma omp ordered
    statement_block
```

Notes
The **omp ordered** directive must be used as follows:

- It must appear within the extent of a **omp for** or **omp parallel for** construct containing an **ordered** clause.
- It applies to the statement block immediately following it. Statements in that block are executed in the same order in which iterations are executed in a sequential loop.
- An iteration of a loop must not execute the same **omp ordered** directive more than once.
- An iteration of a loop must not execute more than one distinct **omp ordered** directive.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"#pragma omp for Preprocessor Directive (C Only)" on page 253
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66

#pragma omp parallel Preprocessor Directive (C Only)

The **omp parallel** directive explicitly instructs the compiler to parallelize the chosen segment of code.

When a parallel region is encountered, a logical team of threads is formed. Each thread in the team executes all statements within a parallel region except for work-sharing constructs. Work within work-sharing constructs is distributed among the threads in a team.

Syntax

```
#pragma omp parallel [clause[ clause] ...]
<statement_block>
```

where *clause* is any of the following:

if (*exp*)	When the **if** argument is specified, the program code executes in parallel only if the scalar expression represented by *exp* evaluates to a non-zero value at run-time. Only one **if** clause can be specified.
private (*list*)	Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.
firstprivate (*list*)	Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.
shared (*list*)	Declares the scope of the data variables in *list* to be shared across all threads.

default (shared | none)
: Defines the default data scope of variables in each thread. Only one **default** clause can be specified on an **omp parallel** directive.

  Specifying **default(shared)** is equivalent to stating each variable in a **shared(***list***)** clause.

  Specifying **default(none)** requires that each data variable visible to the parallelized statement block must be explcitly listed in a data scope clause, with the exception of those variables that are:
  - const-qualified,
  - specified in an enclosed data scope attribute clause, or,
  - used as a loop control variable referenced only by a corresponding **omp for** or **omp parallel for** directive.

copyin (*list*)
: For each data variable specified in *list*, the value of the data variable in the master thread is copied to the thread-private copies at the beginning of the parallel region. Data variables in *list* are separated by commas.

  Each data variable specified in the **copyin** clause must be a **threadprivate** variable.

reduction (*operator*: *list*)
: Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

  A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

  Variables specified in the **reduction** clause:
  - must be of a type appropriate to the operator.
  - must be shared in the enclosing context.
  - must not be const-qualified.
  - must not have pointer type.

Notes

Loop iterations must be independent before the loop can be parallelized. An implied barrier exists at the end of a parallelized statement block.

Nested parallel regions are always serialized.

RELATED CONCEPTS

Program Parallelization
Shared and Private Variables in a Parallel Environment

RELATED TASKS

Control Parallel Processing with Pragmas

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66
"#pragma omp parallel sections Preprocessor Directive (C Only)" on page 260

#pragma omp parallel for Preprocessor Directive (C Only)

The **omp parallel for** directive effectively combines the **omp parallel** and **omp for** directives. This directive lets you define a parallel region containing a single **for** directive in one step.

Syntax

```
#pragma omp parallel for [clause[ clause] ...]
<for_loop>
```

All clauses and restrictions described in the **omp parallel** and **omp for** directives apply to the **omp parallel for** directive.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66
"#pragma omp parallel Preprocessor Directive (C Only)" on page 258
"#pragma omp for Preprocessor Directive (C Only)" on page 253

#pragma omp parallel sections Preprocessor Directive (C Only)

The **omp parallel sections** directive effectively combines the **omp parallel** and **omp sections** directives. This directive lets you define a parallel region containing a single **sections** directive in one step.

Syntax

```
#pragma omp parallel sections [clause[ clause] ...]
   {
     [#pragma omp section]
          statement-block
     [#pragma omp section]
          statement-block
       .
       .
       .
     ]
   }
```

All clauses and restrictions described in the **omp parallel** and **omp sections** directives apply to the **omp parallel sections** directive.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60

"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66
"#pragma omp parallel Preprocessor Directive (C Only)" on page 258
"#pragma omp sections Preprocessor Directive (C Only)"

#pragma omp sections Preprocessor Directive (C Only)

The **omp sections** directive distributes work among threads bound to a defined parallel region.

Syntax

```
#pragma omp sections [clause[ clause] ...]
   {
     [#pragma omp section]
          statement-block
     [#pragma omp section]
          statement-block
       .
       .
       .
   }
```

where *clause* is any of the following:

private (*list*)	Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.
firstprivate (*list*)	Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.
lastprivate (*list*)	Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last **section**. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.
reduction (*operator*: *list*)	Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas. A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable. Variables specified in the **reduction** clause: • must be of a type appropriate to the operator. • must be shared in the enclosing context. • must not be const-qualified. • must not have pointer type.
nowait	Use this clause to avoid the implied **barrier** at the end of the **sections** directive. This is useful if you have multiple independent work-sharing sections within a given parallel region. Only one **nowait** clause can appear on a given **sections** directive.

Notes

The **omp section** directive is optional for the first program code segment inside the **omp sections** directive. Following segments must be preceded by an **omp section** directive. All **omp section** directives must appear within the lexical construct of the program source code segment associated with the **omp sections** directive.

When program execution reaches a **omp sections** directive, program segments defined by the following **omp section** directive are distributed for parallel execution among available threads. A barrier is implicitly defined at the end of the larger program region associated with the **omp sections** directive unless the **nowait** clause is specified.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66
"#pragma omp parallel sections Preprocessor Directive (C Only)" on page 260

#pragma omp single Preprocessor Directive (C Only)

The **omp single** directive identifies a section of code that must be run by a single available thread.

Syntax

```
#pragma omp single [clause[ clause] ...]
   statement_block
```

where *clause* is any of the following:

private (*list*)	Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.
firstprivate (*list*)	Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.
nowait	Use this clause to avoid the implied **barrier** at the end of the **single** directive. Only one **nowait** clause can appear on a given **single** directive.

Notes

An implied barrier exists at the end of a parallelized statement block unless the **nowait** clause is specified.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66

#pragma omp threadprivate Preprocessor Directive (C Only)

The **omp threadprivate** directive defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.

Syntax

```
#pragma omp threadprivate (list)
```

where *list* is a comma-separated list of variables.

Notes
Each copy of an **omp threadprivate** data variable is initialized once prior to first use of that copy. If an object is changed before being used to initialize a **threadprivate** data variable, behavior is unspecified.

A thread must not reference another thread's copy of an **omp threadprivate** data variable. References will always be to the master thread's copy of the data variable when executing serial and master regions of the program.

Use of the **omp threadprivate** directive is governed by the following points:

- An **omp threadprivate** directive must appear at file scope outside of any definition or declaration.
- A data variable must be declared with file scope prior to inclusion in an **omp threadprivate** directive *list*.
- An **omp threadprivate** directive and its *list* must lexically precede any reference to a data variable found in that *list*.
- A data variable specified in an **omp threadprivate** directive in one translation unit must also be specified as such in all other translation units in which it is declared.
- Data variables specified in an **omp threadprivate** *list* must not appear in any clause other than the **copyin**, **schedule**, and **if** clauses.
- The address of a data variable in an **omp threadprivate** *list* is not an address constant.
- A data variable specified in an **omp threadprivate** *list* must not have an incomplete or reference type.

RELATED CONCEPTS

"Chapter 2. Program Parallelization (C Only)" on page 53
"Shared and Private Variables in a Parallel Environment (C Only)" on page 57
"Countable Loops (C Only)" on page 54

RELATED TASKS

"Control Parallel Processing with Pragmas (C Only)" on page 59

RELATED REFERENCES

"#pragma Preprocessor Directives for Parallel Processing (C Only)" on page 60
"OpenMP Run-time Options for Parallel Processing (C Only)" on page 66

Contacting IBM

We're delighted to offer a solid cornerstone for your application development: IBM's comprehensive support services. Whether you are an occasional user with a simple question, a power user with many complex technical questions, or someone who requires application design assistance and consulting, IBM Support can meet your needs.

Comments on This Help

Please let us know about any errors or omissions in this online help or in the hardcopy Getting Started book, or our PDF documents. Send your e-mail to: torrcf@ca.ibm.com

Fee Support

Developers on the VisualAge C++ for AIX Service and Support team handle everything from simple how-to's to complex technical problems. Solutions may take the form of a brief explanation, a workaround, a fix to the current product, or a fix to the next release.

http://www.ibm.com/support/ describes IBM Support Offerings on all platforms, worldwide.

http://www.ibm.com/rs6000/support/ describes support offerings on the RS/6000 platform, in your country. It also indicates whether your country provides support electronically over the Internet in addition to telephone support.

http://www.lotus.com/passport describes the IBM and Lotus® Passport Advantage™ contracting option.

The IBM Software Support Handbook, accessible from http://www.ibm.com/software/support, also lists worldwide support contacts.

Phone numbers for information on Support Line offerings are:

- **United States**: 1-888-426-4343 (IBM Global Services), option 3 as of December 1999. Should this number change, IBM general information at 1-800-IBM-4YOU (1-800-426-4968) can route you to the appropriate group.
- **Canada**: 1-800-465-9600, option 3 as of December 1999. Should this number change, you can also contact IBM general information at 1-800-IBM-4YOU (1-800-426-4968).

Please call 1-800-266-8720 in the U.S. and Canada for information on Passport Advantage offerings.

- Elsewhere, please contact your local IBM office.

If you contact support, please have the following information available:

- The product name
- The product version
- The hardware configuration and software (product names and versions you are using)
- What happened and what you were doing when the problem occurred

- Whether you tried to solve the problem and how
- The exact wording of any messages displayed

Consulting Services

VisualAge and WebSphere™ Product Affinity Services Group is a core group of technical specialists from the IBM development labs that created the IBM VisualAge and WebSphere products. With access to a network of IBM product area experts, IBM and industry business partners, and some of the best resources in the industry, we can put the optimal team in place to meet the challenge of absorbing new technology. Our goal is to enable organizational success with VisualAge and WebSphere — ensuring that our products are used effectively within your development team.

For more information, visit http://www.ibm.com/software/ad/vaws-services/ or contact the Product Affinity Services Team at:

AIM_SERVICES@us.ibm.com