

XL C/C++ Advanced Edition for Mac OS X



# Programming Tasks

*Version 6.0*



XL C/C++ Advanced Edition for Mac OS X



# Programming Tasks

*Version 6.0*

**Note!**

Before using this information and the product it supports, read the information in "Notices" on page 35.

**First Edition (December 2003)**

This edition applies to version 6.0 of XL C/C++ Advanced Edition for Mac OS X (product number 5724-G12) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. You can send them to [compinfo@ca.ibm.com](mailto:compinfo@ca.ibm.com). Be sure to include your e-mail address if you want a reply. Include the title and order number of this book, and the page number or topic related to your comment.

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

© Copyright International Business Machines Corporation 1998, 2003. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

## Chapter 1. Program Stream I/O . . . . . 1

Standard Streams . . . . .	1
File Handles for Standard Streams . . . . .	2
Redirecting Standard Streams. . . . .	2

## Chapter 2. Data Mapping and Storage . . . 3

Default Alignment of Aggregates . . . . .	4
Alignment of Bit Fields . . . . .	5
Alignment Examples. . . . .	6
Storage of Floating Point Data . . . . .	8
Storage of float and double Types . . . . .	9
Storage of int, long, and short Types . . . . .	9
Storage of Vector Types . . . . .	10
Mapping of Automatic Variables . . . . .	11
Mapping of Compound Data Types . . . . .	11

## Chapter 3. Signals and Exception Handling . . . . . 13

Signals . . . . .	13
Signal Handling . . . . .	13
Program Signal Handling. . . . .	14
Example of Using Volatile Variables . . . . .	14

## Chapter 4. Optimization . . . . . 17

Optimization Techniques Used by XL C/C++ . . . . .	18
--	----

Coding Your Application to Improve Performance . . . . .	19
Find Faster I/O Techniques . . . . .	19
Reduce Function-Call Overhead . . . . .	20
Manage Memory Efficiently . . . . .	21
Optimize Variables . . . . .	21
Manipulate Strings Efficiently . . . . .	22
Optimize Expressions and Program Logic . . . . .	22

## Chapter 5. Floating Point Operations . . . 25

Compile-Time Floating-Point Arithmetic. . . . .	25
Rounding Mode Restrictions. . . . .	26

## Chapter 6. Constructing a Library . . . . . 27

Initialize Shared Library (C++) . . . . .	27
Specify Priority Levels for Library Objects . . . . .	29
Example of Object Initialization in a Group of Files (C++) . . . . .	30

## Chapter 7. Framework Header Files . . . 33

## Notices . . . . . 35

Programming Interface Information . . . . .	36
Trademarks and Service Marks . . . . .	37
Industry Standards . . . . .	37



---

## Chapter 1. Program Stream I/O

Input and output are mapped into logical data streams, either text or binary. Streams present a consistent view of file contents, independent of the underlying file system. I/O can be buffered to increase the efficiency of system level I/O.

### Text Streams

Text streams contain printable characters and control characters.

There may not be an exact correspondence between the characters in a stream and the output. The IBM® XL C/C++ Advanced Edition for Mac OS X compiler may add, alter, or ignore some new-line characters during input or output so that they conform to the conventions for representing text in the operating system environment. Printable characters are not changed.

On output, each new-line character is translated into a carriage-return character, followed by a line-feed character. On input, a line-feed character or a carriage-return character followed by a line-feed character is converted to a new-line character.

### Binary Streams

A binary stream is a sequence of characters or data. The data is not altered on input or output.

---

## Standard Streams

XL C/C++ supports the C standard streams and C++ iostreams.

### C Standard Streams

Any program that includes the header `stdio.h` can use the C standard streams for I/O. The following streams are automatically set up by the run-time environment:

- stdin** The input device from which your application normally retrieves its data. For example, the library function **getchar** uses **stdin**.
- stdout** The output device to which your application normally directs its output. For example, the library function **printf** uses **stdout**.
- stderr** The output device to which your application directs its diagnostic messages.

### C++ iostreams

XL C/C++ Advanced Edition for Mac OS X uses the `iostream` provided by the `gcc` compiler.

The input streams are `istream` and `wistream` objects. The output streams have type `ostream` and `wostream`. The names of the wide character streams and classes start with a “w”. Depending on the level of `g++`, wide character streams may or may not be supported.

The iostream standard stream objects are:

**cin and wcin**

The standard narrow- and wide-character input streams.

**cout and wcout**

The standard narrow- and wide-character output streams.

**cerr and wcerr**

The standard error streams. Output to these streams is unit-buffered. Characters sent to these streams are flushed after each insertion operation.

**clog and wclog**

Additional standard error streams. Output to these streams is fully buffered.

---

## File Handles for Standard Streams

The operating system associates a file handle with each of the streams as follows:

File Handle	C Stream	C++ Stream
0	stdin	cin and wcin
1	stdout	cout and wcout
2	stderr	cerr, clog, wcerr, and wclog

The file handle and stream are not equivalent. There may be situations where a file handle is associated with a different stream. For example, file handle 2 may be associated with a stream other than stderr, cerr, or clog.

---

## Redirecting Standard Streams

By default, the standard streams read from the keyboard and write to the screen. You can redirect the standard streams in the following ways:

**From within an application**

To redirect C standard streams from within your application, use the freopen library function. For example, to send your output to a file called pia.out instead of sending it to stdout, code the following statement in your program:

```
freopen("pia.out", "w", stdout);
```

**From the invocation command on the command line**

The user can redirect C or C++ standard streams when invoking your application from the command line. The user specifies the standard redirection symbols > and < with the file handles for standard streams.

For example, the following command runs the program bill.exe, which has two required parameters XYZ and 123, and redirects the output from stdout to a file called bill.out:

```
bill XYZ 123 > bill.out
```

The user can also redirect one standard stream to another. For example, the following bash shell command redirects stderr to stdout:

```
2> &1
```

## Chapter 2. Data Mapping and Storage

Within a structure, each data type supported by XL C/C++ is mapped into storage with a specific alignment. This alignment depends on the value of the **-qalign** compiler option or **#pragma align**. You can also change the alignment by using the **\_\_align** specifier or the **aligned** variable attribute.

► **Mac OS X** You can specify the following alignment values:

- **-qalign=natural** (Natural column in the table below)
- **-qalign=power** (Power column in the table below) This is the default.
- **-qalign=mac68k** (Mac 68K column in the table below)
- **-qalign=bit\_packed** (Bit Packed column in the table below)

► **Linux** You can specify the following alignment values:

- **-qalign=linuxppc**. This is the default.
- **-qalign=bit\_packed**

► **AIX** You can specify the following alignment values:

- **-qalign=natural**
- **-qalign=full** or **-qalign=power**, which are equivalent. This is the default.
- **-qalign=mac68k** or **-qalign=twobyte**, which are equivalent.
- **-qalign=packed** or **-qalign=bit\_packed**

### Notes:

1. The value of **-qalign** affects only the alignment of members, not the amount of storage used for each member.
2. The alignment given by **-qalign=bit\_packed** is the same on every platform.
3. The following types of alignment have slightly different implementations on the different platforms: **-qalign=natural**, **-qalign=power**, and **-qalign=mac68k**.
4. If you generate data with an application on one platform and read the data with an application on another platform, the data may have an alignment that the reading application does not expect. To avoid this problem, use a platform-neutral mechanism for the layout of data in structures. For example, if you wrap a structure with **#pragma pack** the alignment will be the same on all platforms.

Table 1. Alignment values that are supported on Mac OS X

Type	Storage	Alignment			
		Natural	Power	Mac 68K	Bit Packed
_Bool (C), bool (C++)	4 bytes	4 bytes	4 bytes	2 bytes	1 bytes
char, signed char, unsigned char	1 byte	1 byte	1 byte	1 byte	1 byte
int, unsigned int	4 bytes	4 bytes	4 bytes	2 bytes	1 byte
short int, unsigned short int	2 bytes	2 bytes	2 bytes	2 bytes	1 byte
long int, unsigned long int	4 bytes	4 bytes	4 bytes	2 bytes	1 byte
long long	8 bytes	8 bytes	(see note 1)	2 bytes	1 byte

Table 1. Alignment values that are supported on Mac OS X (continued)

Type	Storage	Alignment			
		Natural	Power	Mac 68K	Bit Packed
float	4 bytes	4 bytes	4 bytes	2 bytes	1 byte
float _Complex	8 bytes	4 bytes	4 bytes	2 bytes	1 byte
double	8 bytes	8 bytes	(see note 1)	2 bytes	1 byte
double _Complex	16 bytes	8 bytes	(see note 1)	2 bytes	1 byte
long double	8 bytes	8 bytes	(see note 1)	2 bytes	1 byte
long double _Complex	16 bytes	8 bytes	(see note 1)	2 bytes	1 byte
pointer	4 bytes	4 bytes	4 bytes	2 bytes	1 byte
vector types	16 bytes	16 bytes	16 bytes	16 bytes	1 byte

**Notes:**

1. For Power alignment, a long long, double, double \_Complex, long double, or long double \_Complex member is 8-byte aligned if it is the first member; otherwise, it is 4-byte aligned.

## Default Alignment of Aggregates

Alignment within a structure can be changed with any of the following:

- **#pragma align**
- **#pragma pack**
- The **\_\_align()** specifier
- The **\_\_attribute\_\_((aligned))** specifier
- The **\_\_attribute\_\_((packed))** specifier
- The **-qalign** compiler option

### Power Alignment

- Vector type members have an alignment of 16 bytes.
- The first element has its natural alignment.
- Subsequent members (other than Vector types) have their natural alignment or 4 bytes, whichever is less.
- The alignment of a structure is the largest alignment value of its members.
- The size of a structure is the smallest multiple of its alignment value that can encompass all of the members of the structure.

### Natural Alignment

- All elements have their natural alignment.
- The alignment of a structure is the largest alignment value of its members.
- The size of a structure is the smallest multiple of its alignment value that can encompass all of the members of the structure.

### Mac 68K Alignment

- Vector type members have an alignment of 16 bytes.
- Char type members have an alignment of 1 byte.
- All elements other than Vector and Char types have an alignment of 2 bytes.
- The alignment of a structure is the largest alignment value of its members or 2 bytes, whichever is greater.

- The size of a structure is the smallest multiple of its alignment value that can encompass all of the members of the structure.

### Bit\_Packed Alignment

- All elements have an alignment of 1 byte.
- The alignment of a structure is the largest alignment value of its members, after the preceding alignment rule has been applied and any alignment modifiers have had effect.
- The size of a structure is the smallest multiple of its alignment value that can encompass all of the members of the structure.

### Nested Aggregates

Aggregates with different alignments can be nested. Each aggregate is laid out using the alignment rules applicable to it. The start position of the nested aggregate is determined by the alignment mode that is in effect when the nested aggregate is declared.

### Vector Types

Vector types are 16-byte aligned. You can override this behavior in any of the following ways:

- Specify **#pragma pack** with a value less than 16.
- Specify **#pragma align(bit\_packed)** or **-qalign=bit\_packed**.
- Specify **\_\_attribute\_\_((packed))**.

The **\_\_align()** and **\_\_attribute\_\_((aligned))** specifiers do not change the alignment of vector types.

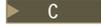
## Alignment of Bit Fields

The following rules apply when bit-field members are mapped out in an aggregate.

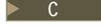
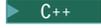
### Power Alignment

- A bit field can be declared as `_Bool` (C), `bool` (C++), `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long` data type.
-  The maximum size of a bit field is the size of its base declared type.
- A zero-length bit field pads to the next alignment boundary of its base declared type. This causes the next member to begin on a byte boundary (for `char` bit fields), 2-byte boundary (for `short`), 4-byte boundary (for `int` or `long`), or 8-byte boundary (for `long long`). Padding does not occur if the previous member's memory layout ended on the appropriate boundary.
-  An aggregate that contains only zero-length bit fields has a length of zero bytes and the alignment of the base declared type of the first member (1 byte for `char`, 2 bytes for `short`, 4 bytes for `int` or `long`, and 8 bytes for `long long`).
-  An aggregate that contains only zero-length bit fields has the length of the base declared type of the first member (1 byte for `char`, 2 bytes for `short`, 4 bytes for `int` or `long`, and 8 bytes for `long long`).

### Natural Alignment

- A bit field can be declared as `_Bool` (C), `bool` (C++), `char`, signed `char`, unsigned `char`, `short`, unsigned `short`, `int`, unsigned `int`, `long`, unsigned `long`, `long long`, or unsigned `long long` data type.
-  The maximum size of a bit field is the size of its base declared type.
- A zero-length bit field pads to the next alignment boundary of its base declared type. This causes the next member to begin on a byte boundary (for `char` bit fields), 2-byte boundary (for `short`), 4-byte boundary (for `int` or `long`), or 8-byte boundary (for `long long`). Padding does not occur if the previous member's memory layout ended on the appropriate boundary.
-  An aggregate that contains only zero-length bit fields has a length of zero bytes and an alignment of 1 byte.
-  An aggregate that contains only zero-length bit fields has a length of 1 byte.

### Mac 68K Alignment

- A bit field can be declared as `_Bool` (C), `bool` (C++), `char`, signed `char`, unsigned `char`, `short`, unsigned `short`, `int`, unsigned `int`, `long`, unsigned `long`, `long long`, or unsigned `long long` data type.
-  The maximum size of a bit field is the size of its base declared type.
- Bit fields are bit packed, and have an alignment of 1 bit.
- A zero-length bit field pads to the next alignment boundary of its base declared type. This causes the next member to begin on a byte boundary (for `char` bit fields), 2-byte boundary (for `short`), 4-byte boundary (for `int` or `long`), or 8-byte boundary (for `long long`). Padding does not occur if the previous member's memory layout ended on the appropriate boundary.
-  An aggregate that contains only zero-length bit fields has a length of zero and an alignment of 2 bytes.
-  An aggregate that contains only zero-length bit fields has a length of 2 bytes and an alignment of 2 bytes.

### Bit\_Packed Alignment

- A bit field can be declared as `_Bool` (C), `bool` (C++), `char`, signed `char`, unsigned `char`, `short`, unsigned `short`, `int`, unsigned `int`, `long`, unsigned `long`, `long long`, or unsigned `long long` data type.
-  The maximum size of a bit field is the size of its base declared type.
- Bit fields have an alignment of 1 bit, and are packed with no default padding between bit fields.
- A zero-length bit field has no effect on the alignment of the next member.

---

## Alignment Examples

The following examples use these symbols to show padding and boundaries:

```
p = padding
| = halfword (2-byte) boundary
: = byte boundary
```

### Mac 68K Example

For:

```

#pragma options align=mac68k
struct B {
    char a;
    double b;
}
#pragma options align=reset

sizeof(B) == 10
alignof(B) == 2

```

The layout of B is:

```
|a:p|b:b|b:b|b:b|b:b|
```

### Bit Packed Example

For:

```

#pragma options align=bit_packed
struct {
    int a : 8;
    int b : 10;
    int c : 12;
    int d : 4;
    int e : 3;
    int : 0;
    int f : 1;
    char g;
} A;
#pragma options align=reset

sizeof(A) == 7
alignof(A) == 1

```

The layout of A is:

Member Name	Displacement Bytes (Bits)
a	0
b	1
c	2 (2)
d	3 (6)
e	4 (2)
f	5
g	6

### Nested Aggregate Example

The following example uses these symbols to show padding and boundaries:

```

p = padding
| = halfword (2-byte) boundary
: = byte boundary

```

For:

```

#pragma options align=mac68k
struct A {
    char a;
    #pragma options align=power
    struct B {
        int b;

```

```

        char c;
    } B1;    // <-- B1 laid out using Power alignment rules
#pragma options align=reset    // <-- has no effect on A or B, but
                                //    on subsequent structs

    char d;
};
#pragma options align=reset

sizeof(A) == 12
alignof(A) == 2

```

The layout of A is:

```
|a:p|b:b|b:b|c:p|p:p|d:p|
```

## Storage of Floating Point Data

XL C/C++ conforms to IEEE format, in which a floating point number is represented in terms of sign (S), exponent (E), and fraction (F):

$$(-1)^S \times 2^E \times 1.F$$

### 4-Byte (float) Data

In the internal representation, there is 1 bit for the sign (S), 8 bits for the exponent (E), and 23 bits for the fraction (F). The bits are mapped with the fraction in bit 0 to bit 22, the exponent in bit 23 to bit 30, and the sign in bit 31:

```

3 32222222 222111111111110000000000
1 09876543 21098765432109876543210
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFF

```

Read the lines vertically from top to bottom. For example, the third column of numbers shows that bit 61 is part of the exponent.

The number is stored as follows, with high memory to the right:

```

byte 0  byte 1  byte 2  byte 3
00000000 11111100 22221111 33222222
76543210 54321098 32109876 10987654
FFFFFFFF FFFFFFFF EFFFFFFF SEEEEEEE

```

### 8-Byte (double) Data

In the internal representation, there is 1 bit for the sign (S), 11 bits for the exponent (E), and 52 bits for the fraction (F). The bits are mapped with the fraction in bit 0 to bit 51, the exponent in bit 52 to bit 62, and the sign in bit 63:

```

6 6665555555 55444444444433333333322222222211111111110000000000
3 21098765432 1098765432109876543210987654321098765432109876543210
S EEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

```

The number is stored as follows, with high memory to the right:

```

byte 0  byte 1  byte 2  ...  byte 5  byte 6  byte 7
00000000 11111100 22221111  ...  44444444 55555544 66665555
76543210 54321098 32109876  ...  76543210 54321098 32109876
FFFFFFFF FFFFFFFF FFFFFFFF  ...  FFFFFFFF EEEEEFFF SEEEEEEE

```

---

## Storage of float and double Types

Specifier	Storage Allocated
float	4 bytes
float _Complex	8 bytes
double	8 bytes
double _Complex	16 bytes
long double	8 bytes

To declare a data object having a floating-point type, use the *float specifier*.

The float specifier has the form:



The declarator for a simple floating-point declaration is an identifier. You can initialize a simple floating-point variable with a float constant or with a variable or expression that evaluates to an integer or floating-point number. The storage class of a variable determines how you initialize the variable.

The following example defines the identifier `pi` as an object of type `double`:

```
double pi;
```

The following example defines the float variable `real_number` with the initial value 100.55:

```
static float real_number = 100.55f;
```

The following example defines the float variable `float_var` with the initial value 0.0143:

```
float float_var = 1.43e-2f;
```

The following example declares the long double variable `maximum`:

```
extern long double maximum;
```

The following example defines the array `table` with 20 elements of type `double`:

```
double table[20];
```

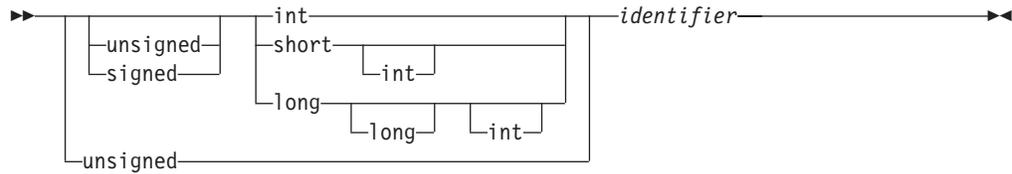
---

## Storage of int, long, and short Types

Specifier	Storage Allocated
short, short int	2 bytes
int	4 bytes
long, long int	4 bytes
long long, long long int	8 bytes

To declare a data object having an integer data type, use an int type specifier.

The **int** specifier has the form:



The declarator for a simple integer definition or declaration is an identifier. You can initialize a simple integer definition with an integer constant or with an expression that evaluates to a value that can be assigned to an integer. The storage class of a variable determines how you can initialize the variable.

The unsigned prefix indicates that the object is a nonnegative integer. Each unsigned type provides the same size storage as its signed equivalent. For example, `int` reserves the same storage as unsigned `int`. Because a signed type reserves a sign bit, an unsigned type can hold a larger positive integer than the equivalent signed type.

The following example defines the short int variable `flag`:

```
short int flag;
```

The following example defines the int variable `result`:

```
int result;
```

The following example defines the unsigned long int variable `ss_number` as having the initial value 438888834:

```
unsigned long ss_number = 438888834ul;
```

The following example defines the identifier `sum` as an object of type `int`. The initial value of `sum` is the result of the expression `a + b`:

```
extern int a, b;
auto sum = a + b;
```

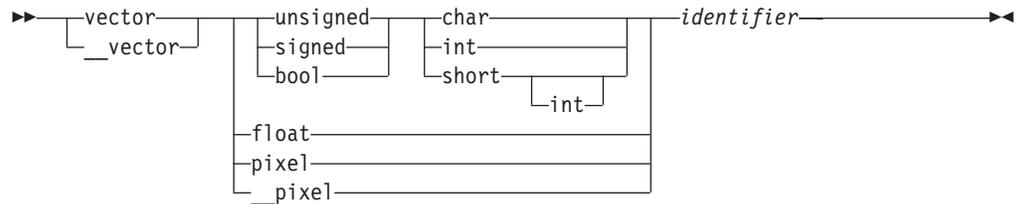
## Storage of Vector Types

XL C/C++ Advanced Edition for Mac OS X supports the following vector types by implementing the AltiVec Programming Interface specification.

Specifier	Interpretation
<code>vector unsigned char</code>	16 unsigned char
<code>vector signed char</code>	16 signed char
<code>vector bool char</code>	16 unsigned char
<code>vector unsigned short</code> , <code>vector unsigned short int</code>	8 unsigned short
<code>vector signed short</code> , <code>vector signed short int</code>	8 signed short
<code>vector bool short</code> , <code>vector bool short int</code>	8 unsigned short
<code>vector unsigned int</code>	4 unsigned int
<code>vector signed int</code>	4 signed int
<code>vector bool int</code>	4 unsigned int

Specifier	Interpretation
vector float	4 float
vector pixel	8 unsigned short

The **vector** specifier has the form:



The following types are also supported, but are deprecated:

- vector unsigned long and vector unsigned long int are equivalent to vector unsigned int
- vector signed long and vector signed long int are equivalent to vector signed int
- vector bool long and vector bool long int are equivalent to vector bool int

---

## Mapping of Automatic Variables

For automatic variables, consider the following information:

- Automatic variables have the same mapping as other variables.
- When optimization is turned on, automatic variables are ordered to minimize padding.
- Automatic variables are always mapped on the stack instead of a data segment. Because memory on the stack is constantly reallocated, automatic variables are not guaranteed to be retained after the return of the function that used them.

---

## Mapping of Compound Data Types

You can access the allocated storage for the following compound data types in C and C++ programs:

- Null-Terminated Character Strings
- Fixed-Length Arrays Containing Simple Data Types
- Aligned Structures
- Unaligned or Packed Structures
- Arrays of Structures

► **C++** The C++ compiler may generate extra fields for classes that contain base classes or virtual functions. Objects of these types may not conform to the usual mappings for structures.



---

## Chapter 3. Signals and Exception Handling

This chapter describes how your program can handle signals and exceptions.

---

### Signals

A signal is a software interrupt. The following types of events raise signals:

- A machine interrupt, such as divide by zero. This is a very common source of signals.
- Your program can send a signal to itself with the **raise** function.
- The shell can generate signals in response to user-defined keystrokes. For example, Ctrl-C is commonly defined as the SIGINT signal. Use the **stty -a** command to determine which signals are set for your shell.
- The operating system may send a signal. For example, SIGSEGV may be sent for an invalid memory reference.

Signals comply with the C and C++ standards. If you want your application to be portable to other operating systems, you can use a signal handler to detect operating system exceptions.

Operating system signals can be either synchronous or asynchronous.

- Synchronous signals are caused by code in the thread that receives the signal. Most operating system signals are synchronous.
- Asynchronous signals are caused by actions outside of your current thread, for example, typing Ctrl-C.

#### C++ Exception Handling

C++ exception constructs such as **try**, **throw** and **catch** exist only within the C++ language. C++ exception handlers cannot intercept operating system exceptions, such as access violations.

---

### Signal Handling

You can handle signals in either of the following ways:

- Accept the default handling provided by XL C/C++, which usually results in program termination with a message.
- Program signal handling.

#### When to Simply Debug

To eliminate signals that you suspect are due to program logic, use a debugger.

Here are some other common problems:

- Improper use of memory. Using a pointer to an object that has already been freed can cause an exception.
- Using an invalid pointer.
- Passing an invalid parameter to a system function.
- Return codes from library or system calls that are not checked.

## When Special Handling is Required

Floating point exceptions and two classes of library functions, math functions and critical functions, require special handling. Operating system signals that occur in all other library functions are treated as though they occurred in regular user code.

If your program links with shared libraries that link to more than one library environment, you must take steps to ensure that the right handler is called.

---

## Program Signal Handling

Use the **signal** function to specify how to handle signals. For each signal, you can specify one of the types of handlers listed below. The signal constants are defined in `<signal.h>`.

**SIG\_DFL** Specifies the default action. This is the initial setting for all signals. For most signals, the default action is to terminate the process with an error message.

**SIG\_IGN** Ignores the condition and tries to continue running the program.

### Your own signal handler function

Registers the function you specify. This can be a function you have written. When the signal is reported and your function is called, signal handling is reset to **SIG\_DFL** to prevent recursion should the same signal be reported from your function.

To reset default handling for a signal, call the signal in a statement similar to the following. Specify the signal name in the first argument of `signal`.

```
signal(name, SIG_DFL);
```

---

## Example of Using Volatile Variables

User variables that are referenced by multiple threads should have the attribute **volatile** to ensure that all changes to the value of the variable are performed immediately by the compiler.

Because of the way the XL C/C++ compiler optimizes code, the following example may not work as intended if it is built with the optimization options.

```
#include <io.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>

void sig_handler(int);
static int stepnum;

int main()
{
    stepnum = 0;
    signal(SIGSEGV, sig_handler);
    /* code omitted - does not use stepnum */
    stepnum = 1;
    /* code omitted - does not use stepnum */
    stepnum = 2;
    return 0;
}

void sig_handler(int x)
{
```

```
char FileData[50];
sprintf(FileData, "Error at Step %d\n\r", stepnum);
write (2, FileData, strlen(fileData));
}
```

An optimized program may not immediately store the value 1 when 1 is assigned to the variable *stepnum*. It may never store the value 1 and only store the value 2. If a signal occurs between the assignments to *stepnum*, the value passed to *sig\_handler* may not be correct.

Declaring a variable (*stepnum*) as **volatile** indicates to the compiler that references to the variable have side effects, or that the variable may change in ways the compiler cannot determine. Optimization will not eliminate any action involving the volatile variable. Changes to the value of the variable are then stored immediately, and uses of the variable will always cause it to be reloaded from memory.



---

## Chapter 4. Optimization

The compiler transforms source code into object code. You can use the compiler's optimization features to produce object code that is faster, smaller, or both. Some optimizations produce code that is both faster and smaller. In other cases, there is a trade-off between speed and size.

In addition to the benefits of optimization, you should also consider the costs. Optimization increases compilation time, increases the space used during compilation, and decreases the usefulness of debugging information.

To take the best advantage of the compiler's optimization features, you should write code that strictly conforms to the appropriate language standard.

### Speed versus Size

To minimize the size of the object code, specify the **-qcompact** compiler option. Using this option may increase execution time.

For larger programs that are not compute-intensive, optimizing for size might result in a faster program than optimizing for speed. Global effects such as improved paging and cache performance may outweigh the local effects of slower instruction sequences.

If both size and speed are important, consider balancing the performance by optimizing some modules for speed, and others for size. Determine which modules contain hotspots, and are compute-intensive; these should be optimized for speed. All other modules should be optimized for size. To find the right balance, you may need to experiment with different combinations of techniques.

### Specific Hardware

If you want to tune your application for a specific subset of the supported systems, you can specify:

- The architecture (**-qarch** option)
- The microprocessor (**-qtune** option)
- The cache or memory geometry (**-qcache** option)

### Exceptions and Stack Unwinding

 If your program does not throw any C++ exceptions, you can use the **-qnoeh** option. This option allows the compiler to omit cleanup code.

If the stack will not be unwound while any routine in this compilation is active, you can use the **-qnounwind** option. This option can improve optimization of non-volatile register saves and restores. In C++, the **-qnounwind** option implies the **-qnoeh** option.

### When to Optimize

Optimize your code throughout your development cycle. Develop, test, and optimize incrementally rather than developing and testing and then optimizing the entire application at the end.

You can use profile-directed feedback to tune the performance of your application for a typical usage scenario. First, compile the program with the `-qpdf1` option. Generate profile data by using the compiled program in the same ways that users will typically use it. Compile the program again, with the `-qpdf2` option. This optimizes the program based on the profile data.

---

## Optimization Techniques Used by XL C/C++

By default, the compiler *does not* optimize your program. To optimize your program, specify the `-qoptimize` option. You can specify optimization level 2, 3, 4, or 5. The optimization level determines which optimization techniques the compiler uses.

The compiler uses the following techniques at optimization level 2 or higher:

- Eliminating common subexpressions that are recalculated in subsequent expressions. For example, with these expressions:

```
a = c + d;  
f = c + d + e;
```

the common expression `c + d` is saved from its first evaluation and is used in the subsequent statement to determine the value of `f`.

- Simplifying algebraic expressions. For example, the compiler combines multiple constants that are used in the same expression.
- Evaluating constants at compile time
- Eliminating unused or redundant code, including:
  - Code that cannot be reached
  - Code whose results are not subsequently used
  - Store instructions whose values are not subsequently used
- Rearranging the program code to minimize branching logic, combine physically separate blocks of code, and minimize execution time
- Allocating variables and expressions to available hardware registers using a graph coloring algorithm
- Replacing less efficient instructions with more efficient ones. For example, in array subscripting, an add instruction replaces a multiply instruction.
- Moving invariant code out of a loop, including:
  - Expressions whose values do not change within the loop
  - Branching code based on a variable whose value does not change within the loop
  - Store instructions
- Unrolling some loops (`-qunroll`)
- Pipelining some loops

The compiler uses the following techniques at optimization level 3 or higher:

- Unrolling deeper loops and improving loop scheduling
- Increasing the scope of optimization
- Performing optimizations with marginal or niche effectiveness, which may not help all programs

- Performing optimizations that are expensive in compile time or space
- Reordering some floating-point computations, which may produce precision differences or affect the generation of floating-point-related exceptions (**-qnostrict**)
- Eliminating implicit memory usage limits (**-qmaxmem=-1**)

The compiler uses the following techniques at optimization level 4 or higher:

- Interprocedural analysis, which invokes the optimizer at link time to perform optimizations across multiple source files (**-qipa**)
- High-order transformations, which provide optimized handling of loop nests and array language constructs (**-qhot**)
- Hardware-specific optimization (**-qarch=auto**, **-qtune=auto**, and **-qcache=auto**)

The compiler uses the following technique at optimization level 5:

- More detailed interprocedural analysis (**-qipa=level=2**)

---

## Coding Your Application to Improve Performance

Before you begin optimizing, you should check your application for the following potential improvements:

- Choose efficient algorithms with small memory footprints
- Avoid duplicate copies of data
- Structure data to minimize padding between items

The following sections contain specific suggestions for improving the performance of your application:

- “Find Faster I/O Techniques”
- “Reduce Function-Call Overhead” on page 20
- “Manage Memory Efficiently” on page 21
- “Optimize Variables” on page 21
- “Manipulate Strings Efficiently” on page 22
- “Optimize Expressions and Program Logic” on page 22

### Find Faster I/O Techniques

There are a number of ways to improve your program’s performance of input and output:

- Use binary streams instead of text streams. In binary streams, data is not changed on input or output.
- Use the low-level I/O functions, such as **open** and **close**. These functions are faster and more specific to the application than the stream I/O functions like **fopen** and **fclose**. You must provide your own buffering for the low-level functions.
- If you do your own I/O buffering, make the buffer a multiple of 4K, which is the size of a page.
- When reading input, read in a whole line at once rather than one character at a time.
- If you know you have to process an entire file, determine the size of the data to be read in, allocate a single buffer to read it to, read the whole file into that buffer at once using **read**, and then process the data in the buffer. This reduces

disk I/O, provided the file is not so big that excessive swapping will occur. Consider using the `mmap` function to access the file.

- Instead of `scanf` and `fscanf`, use `fgets` to read in a string, and then use one of `atoi`, `atol`, `atof`, or `atold` to convert it to the appropriate format.
- Use `sprintf` only for complicated formatting. For simpler formatting, such as string concatenation, use a more specific string function.

## Reduce Function-Call Overhead

When you write a function or call a library function, consider the following suggestions:

- Call a function directly, rather than using function pointers.
- Pass a value to a function as an argument, rather than letting the function take the value from a global variable.
- Use constant arguments in inlined functions whenever possible. Functions with constant arguments provide more opportunities for optimization.
- Use the `#pragma isolated_call` preprocessor directive to list functions that have no side effects and do not depend on side effects.
- Declare a nonmember function as static whenever possible. This may speed up calls to the function.
-  Use virtual functions only when necessary. They are usually compiled to be indirect calls, which are slower than direct calls.
-  Usually, you should not declare virtual functions inline. If all virtual functions in a class are inline, the virtual function table and all the virtual function bodies will be replicated in each compilation unit that uses the class.
-  When declaring functions, use the `const` specifier whenever possible.
-  Fully prototype all functions. A full prototype gives the compiler and optimizer complete information about the types of the parameters. As a result, promotions from unwidened types to widened types are not required, and parameters may be passed in appropriate registers.
-  Avoid using unprototyped variable argument functions.
- Design functions so that the most frequently used parameters are in the left-most positions in the function prototype.
- Avoid passing by value structures or unions as function parameters or returning a structure or a union. Passing such aggregates requires the compiler to copy and store many values. This is worse in C++ programs in which class objects are passed by value because a constructor and destructor are called when the function is called. Instead, pass or return a pointer to the structure or union, or pass it by reference.
- Pass small types such as `int` and `short` by value rather than passing by reference, whenever possible.
- If your function exits by returning the value of another function with the same parameters that were passed to your function, put the parameters in the same order in the function prototypes. The compiler can then branch directly to the other function.
- Use the intrinsic and built-in functions, which include string manipulation, floating-point, and trigonometric functions, instead of coding your own. Intrinsic functions require less overhead and are faster than a function call, and often allow the compiler to perform better optimization.

▶ **C++** Your functions are automatically mapped to intrinsic functions if you include the XL C/C++ header files.

▶ **C** Your functions are mapped to intrinsic functions if you include `<math.h>` and `<string.h>`.

- Selectively mark your functions for inlining, using the **inline** keyword. An inlined function requires less overhead and is generally faster than a function call. The best candidates for inlining are small functions that are called frequently from a few places. You might also want to put these functions into header files. Large functions and functions that are called rarely may not be good candidates for inlining. Be sure to inline all functions that only load or store a value.
- Avoid breaking your program into too many small functions. If you must use small functions, seriously consider using **-qipa**.
- ▶ **C++** Avoid virtual functions and virtual inheritance unless required for class extensibility. These language features are costly in object space and function invocation performance.

## Manage Memory Efficiently

Because C++ objects are often allocated from the heap and have limited scope, memory use in C++ programs affects performance more than in C programs.

- In a structure, declare the largest members first.
- In a structure, place variables near each other if they are frequently used together.
- ▶ **C++** Ensure that objects that are no longer needed are freed or otherwise made available for reuse. One way to do this is to use an *object manager*. Each time you create an instance of an object, pass the pointer to that object to the object manager. The object manager maintains a list of these pointers. To access an object, you can call an object manager member function to return the information to you. The object manager can then manage memory usage and object reuse.
- ▶ **C++** Avoid copying large, complicated objects.
- ▶ **C++** Avoid performing a *deep copy* if a *shallow copy* is all you require. For an object that contains pointers to other objects, a shallow copy copies only the pointers and not the objects to which they point. The result is two objects that point to the same contained object. A deep copy, however, copies the pointers and the objects they point to, as well as any pointers or objects contained within that object, and so on.

## Optimize Variables

Consider the following suggestions:

- Use local variables, preferably automatic variables, as much as possible. The compiler must make several worst-case assumptions about a global variable. For example, if a function uses external variables and also calls external functions, the compiler assumes that every call to an external function could change the value of every external variable. If you know that a global variable is not affected by any function call, and this variable is read several times with function calls interspersed, copy the global variable to a local variable and then use this local variable.

- If you must use global variables, use static variables with file scope rather than external variables whenever possible. In a file with several related functions and static variables, the optimizer can gather and use more information about how the variables are affected.
- If you must use external variables, group external data into structures or arrays whenever it makes sense to do so. All elements of an external structure use the same base address.
- The `#pragma isolated_call` preprocessor directive can improve the run-time performance of optimized code by allowing the compiler to make less pessimistic assumptions about the storage of external and static variables. Isolated\_call functions with constant or loop-invariant parameters may be moved out of loops, and multiple calls with the same parameters may be replaced with a single call.
- Avoid taking the address of a variable. If you use a local variable as a temporary variable and must take its address, avoid reusing the temporary variable. Taking the address of a local variable inhibits optimizations that would otherwise be done on calculations involving that variable.
- Use constants instead of variables where possible. The optimizer will be able to do a better job reducing run-time calculations by doing them at compile-time instead. For instance, if a loop body has a constant number of iterations, use constants in the loop condition to improve optimization; `for (i=0; i<4; i++)` can be better optimized than `for (i=0; i<x; i++)`.
- Use register-sized integers (**long** data type) for scalars. For large arrays of integers, consider using one- or two-byte integers or bit fields.
- Use the smallest floating-point precision appropriate to your computation.

## Manipulate Strings Efficiently

The handling of string operations can affect the performance of your program.

- When you store strings into allocated storage, align the start of the string on an 8-byte boundary.
- Keep track of the length of your strings. If you know the length of a string, you can use **mem** functions instead of **str** functions. For example, **memcpy** is faster than **strcpy** because it does not have to search for the end of the string.
- If you are certain that the source and target do not overlap, use **memcpy** instead of **memmove**.
- When manipulating strings using **mem** functions, faster code will be generated if the *count* parameter is a constant rather than a variable. This is especially true for small count values.
- Make string literals read-only (the default), whenever possible. This improves certain optimization techniques.

You can explicitly set strings to read-only by using `#pragma strings (readonly)` in your source files or `-qro` to avoid changing your source files.

## Optimize Expressions and Program Logic

Consider the following suggestions:

- If components of an expression are used in other expressions, assign the duplicated values to a local variable.
- Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. For example:

```
float array[10];
float x = 1.0;
int i;
```

```

for (i = 0; i < 9; i++) { /* No conversions needed */
    array[i] = array[i]*x;
    x = x + 1.0;
}
for (i = 0; i < 9; i++) /* Multiple conversions needed */
    array[i] = array[i]*i;

```

When you must use mixed-mode arithmetic, code the integer and floating-point arithmetic in separate computations whenever possible.

- Avoid **goto** statements that jump into the middle of loops. Such statements inhibit certain optimizations.
- Improve the predictability of your code by making the fall-through path more probable. Code such as:

```

if (error) {handle error} else {real code}

```

should be written as:

```

if (!error) {real code} else {error}

```

- If one or two cases of a **switch** statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the **switch** statement.
-  Use **try** blocks for exception handling only when necessary because they can inhibit optimization.
- Keep array index expressions as simple as possible.



---

## Chapter 5. Floating Point Operations

Single precision values have an approximate range of  $10(-38)$  to  $10(+38)$ , with about 7 decimal digits of precision. Double precision values have an approximate range of  $10(-308)$  to  $10(+308)$  and precision of about 16 decimal digits.

When results must be converted to single precision, rounding operations are used. A rounding operation produces the correct single-precision value based on the IEEE rounding mode in effect. Because explicit rounding operations are required, single-precision computations are often slower than double-precision computations. On many other machines, the reverse is true: single-precision operations are faster than double-precision operations. Code ported from other systems can show different performance on the PowerPC<sup>®</sup> architecture.

The PowerPC hardware provides both single-precision and double-precision operations that multiply two numbers and add a third number to the product. These multiply-add-fusion (maf) operations are performed in the same time as a multiply or an add operation alone. The maf functions provide an extension to the IEEE standard because they perform the multiply and add with one (rather than two) rounding errors. The maf functions are both faster and more accurate than the equivalent separate operations. Use the **-qfloat=nomaf** option to suppress the generation of these multiply-add instructions for greater compatibility with the accuracy available on other systems.

**Note:** Single-precision instructions are used with single-precision data.

### Detecting Floating-Point Exceptions

A number of floating-point exceptions can be detected by the floating-point hardware: invalid operation, division by zero, overflow, underflow, and inexact. By default, all exceptions are ignored. However, if you use the **-qflttrap** option, any or all of these exceptions can be detected. In addition, when you add suitable support code to your program, program execution can continue after an exception occurs, and you can then modify the results of operations causing exceptions.

---

## Compile-Time Floating-Point Arithmetic

The compiler attempts to perform as much floating-point arithmetic as possible at compile time. Floating-point operations with constant operands are folded, replacing the operation with the result calculated at compile time. When optimization is enabled, more folding might occur than when optimization is not enabled.

The **-qfloat=fold** option controls the rounding mode that is used at compile time. For example, **-qfloat=nofold** suppresses compile-time rounding.

Compile-time floating-point arithmetic can have two effects on program results:

- In specific cases, the result of a computation at compile time might differ slightly from the result that would have been calculated at run time. The reason is that more rounding operations occur at compile time. For example, where a

multiply-add-fused (MAF) operation might be used at run time, separate multiply and add operations might be used at compile time, producing a slightly different result.

- Computations that produce exceptions can be folded to the IEEE result that would have been produced by default in a run-time operation. This would prevent an exception from occurring at run time. When using the **-qfltrap** option, you should consider using the **-qfloat=nofold** option.

In general, code that affects the rounding mode at run time should be compiled with the **-y** option that matches the rounding mode intended at run time. For example, when the following program:

```
int main ()
{
    union uu
    {
        float x;
        int i;
    } u;
    volatile float one, three;

    u.x=1.0/3.0;
    printf("1/3=%8x \n", u.i);

    one=1.0
    three=3.0;
    u.x=one/three;
    printf ("1/3=%8x \n", u.i);
    return 0;
}
```

is compiled using **-yz**, the expression `1.0/3.0` is folded by the compiler at compile time into a double-precision result. This result is then converted to single-precision and then stored in float `u.x`. The **-qfloat=nofold** option can be specified to suppress all compile-time folding of floating-point computations. The **-y** option only affects compile-time rounding of floating-point computations, but does not affect run-time rounding. The code fragment:

```
one = 1.0;
three = 3.0;
x = one/three;
```

is evaluated at run time in single-precision. Here, the default run-time rounding of “round to nearest” is still in effect and takes precedence over the compile-time specification of “round to zero”. The output of this program is:

```
1/3=3eaaaaaa
1/3=3eaaaaab
```

---

## Rounding Mode Restrictions

The floating-point rounding mode can only be changed at the beginning and end of a function. It cannot be changed across a function call, and if it is changed within a function, it must be restored before returning to the calling routine.

---

## Chapter 6. Constructing a Library

### Shared Libraries

You should compile shared libraries with the `-qmkshrobj` compiler option.

```
xlc++ -c foo.c++
xlc++ -qmkshrobj -o libfoo.dylib foo.o
```

### Static Libraries

To construct a static library, compile each file and then use the Mac OS X `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 example.o
```

---

## 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 `-qmkshrobj` compiler option, then construct an archive (.a) library containing several shared and non-shared objects.

### ► Mac OS X Order of Initialization and Termination

The run-time environment initializes the namespace-scope objects within a single compilation unit in the order of their priority number. Priority numbers can range from 101 to 65535. The smallest priority number that you can specify, 101, is initialized first. The largest priority number, 65535, is initialized last. The default priority of static objects is 65535, and can be controlled with `#pragma priority` or the `init_priority()` variable attribute. Objects with the same priority value are initialized in declaration order.

All global objects within a static application are initialized in link order. The first object file listed on the link step is initialized first, followed by the next object specified. The order of initialization within each object obeys the same rules listed above. For example, if application A links to modules B then C, all initializers are run for object B followed by initializers for object C.

Global objects within a dynamic library are initialized only if the program contains a reference to a member of the library. Libraries are initialized based on the order that members are referenced within the program. For example, program A links to dynamic libraries B and C. When program A calls a function defined in library C, all initializers in library C are run. If program A later calls a function defined in library B, all initializer in library B are then run.

When all of the static objects in a compilation unit have been initialized, the functions that have the constructor attribute are run. All of the initializers from one object file are run, followed by functions marked with the constructor attribute from the same object file, then the initializers and functions from the next object file, and so on. The object files are accessed in link order, and the functions within an object file are run in reverse-definition order.

Objects are terminated in the reverse of the initialization order. Before the object is terminated, the destructor functions are run in link order of the object files and reverse-definition order within an object file.

#### ▶ Linux **Order of Initialization and Termination**

The run-time environment initializes the objects in shared libraries in the order of their priority number. Priority numbers can range from 101 to 65535. If you do not specify priority levels, the default priority is 65535.

The smallest priority number that you can specify, 101, is initialized first. The largest priority number, 65535, is initialized last. Objects with the same priority number are initialized in reverse-link order. (The first object file listed on the link step is initialized last.) By default, objects within a compilation unit are initialized in declaration order.

Within a single shared library or the main function, **#pragma priority** controls the initialization order. Shared libraries are initialized based on their link dependencies. For example, if a program links to libraries A and B, and library B links to library C, C will be initialized before B, and A and B will be initialized before the program.

When all of the static objects in a shared library have been initialized, the functions that have the constructor attribute are run. All of the functions from one object file are run, then the functions from the next object file, and so on. The object files are accessed in reverse-link order, and the functions within an object file are run in reverse-definition order.

Objects are terminated starting with the highest priority number (the reverse of the initialization order). Objects with the same priority number are terminated in link order. By default, objects within a compilation unit are terminated in declaration order. After the objects in a shared library have all been terminated, the destructor functions are run in link order of the object files and reverse-definition order within an object file.

#### ▶ AIX **Order of Initialization and Termination**

The run-time environment initializes the objects in shared 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. If you do not specify priority levels, the default priority is 0 (zero).

The smallest priority number that you can specify, -2147482623, is initialized first. The largest priority number, 2147483647, is initialized last. Objects with the same priority number are initialized in random order.

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.

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

---

## Specify Priority Levels for Library Objects

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(2000) //Following objects constructed with priority 2000
...

static struct base A ;

class house B ;
...
#pragma priority(3000) //Following objects constructed with priority 3000
...

class barn C ;
...
#pragma priority(2500) // Error - priority number must be larger
// than preceding number (3000)
...
#pragma priority(4000) //Following objects constructed with priority 4000
...

class garage D ;
...
```

  You can also specify the priority with the **init\_priority()** attribute. This attribute has the same effect as the **#pragma priority** directive.

### Specifying the Priority Level of a File

To specify the priority level of a file, use the **-qpriority** 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, using the batch compiler option `-qpriority=4000`, is equivalent to using `#pragma priority(4000)`.

If there are no `#pragma priority` directives within the file, all objects within the file have the priority specified with `-qpriority=` .

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 `-qpriority` option and subsequent `#pragma priority` directives must have increasing numbers.

## 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. `#pragma priority`
2. 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:

<pre>farm.C #pragma priority(3000) ... class dog A ; class dog B ; ... #pragma priority(6000) ... class cat C ; class cow D ; ... #pragma priority(7000) class mouse E ; ...</pre>	<pre>zoo.C ... class lion K ; #pragma priority(4000) class bear M ; ... #pragma priority(5000) ... class zebra N ; class snake S ; ... #pragma priority(8000) class frog F ; ...</pre>
--	--

Compile `farm.C` and `zoo.C` with `-qpriority=2000`.

Initialization takes place in the following order:

Object	Priority Value	Comment
"lion K"	2000	Takes priority number of file "zoo.o" (2000) (Initialized first)
"dog A"	3000	Takes #PRAGMA PRIORITY(3000) priority.
"dog B"	3000	Follows "dog A"
"bear M"	4000	Next priority number, specified by #PRAGMA PRIORITY(4000)
"zebra N"	5000	Next priority number from #PRAGMA PRIORITY(5000)
"snake S"	5000	Follows with same priority

<b>Object</b>	<b>Priority Value</b>	<b>Comment</b>
"cat C"	6000	Next priority number
"cow D"	6000	Follows with same priority
"mouse E"	7000	Next priority number
"frog F"	8000	Next priority number (Initialized last).



---

## Chapter 7. Framework Header Files

To add a user-defined framework directory to the framework header file search path, use the **-qframeworkdir** compiler option. This option passes the specified path to the link editor's **-F** option.

By default, the compiler will search for a header file in the following locations, listed in order of search priority, until it is found:

1. Ordinary header file locations
2. User-defined framework directories, specified by the **-qframeworkdir** compiler option
3. System-default framework directories, listed in order of priority:
  - a. /Library/Frameworks/
  - b. /Network/Library/Frameworks/
  - c. /System/Library/Frameworks/
4. Subframework directories, if in an umbrella framework

For example, the following option specification will add **my\_dir1**, **my\_dir2**, and **my\_dir3** to the framework header file search path:

```
-qframeworkdir=my_dir1 -qframeworkdir=my_dir2 -qframeworkdir=my_dir3
```

User-defined framework directories are searched in the order that they are defined to the compiler. In the above example, **my\_dir1** would be searched first, followed by **my\_dir2**, and then **my\_dir3**.

To specify the name of the framework, use the **-framework** compiler option in the following format:

```
-framework framework_name[.extension]
```

For example, you can use the following invocation command to link to the Carbon framework:

```
xlc++ -framework Carbon -o myprogram myprogram.c
```



---

## Notices

This information was developed for products and services offered in the U.S.A.

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

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

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

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

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

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:**

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

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

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

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director  
IBM Canada Ltd. Laboratory  
B3/KB7/8200/MKM  
8200 Warden Avenue  
Markham, Ontario L6G 1C7  
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

---

## 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.

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

---

## Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

- AIX
- IBM
- PowerPC

Other company, product, and service names may be trademarks or service marks of others.

---

## Industry Standards

The following standards are supported:

- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1999 (E)).
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).







Program Number: 5724-G12

SC09-7867-00

