XL C/C++ Advanced Edition V7.0 for Linux

Programming Guide

Version 7.0

SC09-7943-01

XL C/C++ Advanced Edition V7.0 for Linux

Programming Guide

Version 7.0

SC09-7943-01

Note!
Before using this information and the product it supports, read the information in "Notices" on page 47.

Second Edition (March, 2005)

This edition applies to version 7.0.1 of XL C/C++ Advanced Edition V7.0 for Linux (product number 5724-K77) 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. 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.

Contents

About this guide

This guide discusses advanced topics related to the use of the IBM® XL C/C++ Advanced Edition V7.0 for Linux® compiler, with a particular focus on program portability and optimization. The guide provides both reference information and practical tips for getting the most out of the compiler's capabilities, through recommended programming practices and compilation procedures. The guide also contains extensive cross-references to the relevant sections of the other reference guides in the XL C/C++ Advanced Edition V7.0 for Linux documentation set.

This guide includes these topics:

- Chapter 1, "Using 32-bit and 64-bit modes," on page 1 discusses common problems that arise when porting existing 32-bit applications to 64-bit mode, and provides recommendations for avoiding these problems.
- Chapter 2, "Aligning data in aggregates," on page 5 discusses the different compiler options available for controlling the alignment of data in aggregates, such as structures and classes, on all platforms.
- Chapter 3, "Handling floating point operations," on page 11 discusses options available for controlling the way floating-point operations are handled by the compiler.
- Chapter 4, "Using C++ templates," on page 15 discusses the different options for compiling programs that include C++ templates.
- Chapter 5, "Constructing a library," on page 21 discusses how to compile and link static and shared libraries, and how to specify the initialization order of static objects in C++ programs.
- Chapter 6, "Optimizing your applications," on page 27 discusses the various options provided by the compiler for optimizing your programs, and provides recommendations for use of the different options.
- Chapter 7, "Coding your application to improve performance," on page 41 discusses recommended programming practices and coding techniques for enhancing program performance and compatibility with the compiler's optimization capabilities.

Document conventions

Highlighting conventions

This guide uses the following highlighting conventions:

Bold	Identifies commands, keywords, file, directory, and path names, environment variables, executable names, and other items whose names are predefined by the system.
Italics	Identify parameters whose actual names or values are to be supplied by the programmer. *Italics* are also used for the first mention of new terms.
Monospace	Identifies examples of program code.

Examples are intended to be instructional and do not attempt to minimize run time, conserve storage, or check for errors. The examples do not demonstrate all of

the possible uses of language constructs. Some examples are only code fragments and will not compile without additional code.

Icons

In general, this guide documents XL C/C++ functionality as it has been implemented on the Linux platform. However, where issues are discussed that affect portability to other platforms, the following icons are used:

Indicates the functionality supported on the AIX® platform.

Indicates the functionality supported on the Linux platform.

Indicates the functionality supported on the Mac OS X platform.

Indicates a feature that is supported only in the C++ language.

Indicates a feature that is supported only in the C language.

Chapter 1. Using 32-bit and 64-bit modes

You can use XL C/C++ to develop both 32-bit and 64-bit applications. To do so, specify **-q32** (the default) or **-q64**, respectively, during compilation.

However, porting existing applications from 32-bit to 64-bit mode can lead to a number of problems, mostly related to the differences in C/C++ long and pointer data type sizes and alignment between the two modes. The following table summarizes these differences.

Table 1. Size and alignment of data types in 32-bit and 64-bit modes

Data type	32-bit mode		64-bit mode	
	Size	**Alignment**	**Size**	**Alignment**
long, unsigned long	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
pointer	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
size_t (system-defined unsigned long)	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
ptrdiff_t (system-defined long)	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries

The following sections discuss some of the common pitfalls implied by these differences, as well as recommended programming practices to help you avoid most of these issues:

- "Assigning long values" on page 2
- "Assigning pointers" on page 3
- "Aligning aggregate data" on page 4
- "Calling Fortran code" on page 4

When compiling in 32-bit or 64-bit mode, you can use the **-qwarn64** option to help diagnose some issues related to porting applications. In either mode, the compiler immediately issues a warning if undesirable results, such as truncation or data loss, have occurred.

For suggestions on improving performance in 64-bit mode, see "Optimize operations in 64-bit mode" on page 45.

Related references

- **-q32/-q64** in *XL C/C++ Compiler Reference*
- **-qwarn64** in *XL C/C++ Compiler Reference*

Assigning long values

The limits of **long** type integers defined in the **limits.h** standard library header file are different in 32-bit and 64-bit modes, as shown in the following table.

Table 2. Constant limits of long integers in 32-bit and 64-bit modes

Symbolic constant	Mode	Value	Hexadecimal	Decimal
LONG_MIN (smallest signed long)	32-bit	$-(2^{31})$	0x80000000L	–2,147,483,648
	64-bit	$-(2^{63})$	0x8000000000000000L	–9,223,372,036,854,775,808
LONG_MAX (longest signed long)	32-bit	$2^{31}-1$	0x7FFFFFFFL	+2,147,483,647
	64-bit	$2^{63}-1$	0x7FFFFFFFFFFFFFFFL	+9,223,372,036,854,775,807
ULONG_MAX (longest unsigned long)	32-bit	$2^{32}-1$	0xFFFFFFFFUL	+4,294,967,295
	64-bit	$2^{64}-1$	0xFFFFFFFFFFFFFFFFUL	+18,446,744,073,709,551,615

Implications of these differences are:

- Assigning a long value to a **double** variable can cause loss of accuracy.
- Assigning constant values to long-type variables can lead to unexpected results. This issue is explored in more detail in "Assigning constant values to long variables."
- Bit-shifting long values will produce different results, as described in "Bit-shifting long values" on page 3.
- Using **int** and **long** types interchangeably in expressions will lead to implicit conversion through promotions, demotions, assignments, and argument passing, and can result in truncation of significant digits, sign shifting, or unexpected results, without warning.

In situations where a long-type value can overflow when assigned to other variables or passed to functions, you must:

- Avoid implicit type conversion by using explicit type casting to change types.
- Ensure that all functions that return long types are properly prototyped.
- Ensure that long parameters can be accepted by the functions to which they are being passed.

Assigning constant values to long variables

Although type identification of constants follows explicit rules in C and C++, many programs use hexadecimal or unsuffixed constants as ″typeless″ variables and rely on a two's complement representation to exceed the limits permitted on a 32-bit system. As these large values are likely to be extended into a 64-bit **long** type in 64-bit mode, unexpected results can occur, generally at boundary areas such as:

- constant >= **UINT_MAX**
- constant < **INT_MIN**
- constant > **INT_MAX**

Some examples of unexpected boundary side effects are listed in the following table.

Table 3. Unexpected boundary results of constants assigned to long types

Constant assigned to long	Equivalent value	32 bit mode	64 bit mode
–2,147,483,649	INT_MIN–1	+2,147,483,647	–2,147,483,649
+2,147,483,648	INT_MAX+1	–2,147,483,648	+2,147,483,648
+4,294,967,726	UINT_MAX+1	0	+4,294,967,296
0xFFFFFFFF	UINT_MAX	–1	+4,294,967,295
0x100000000	UINT_MAX+1	0	+4,294,967,296
0xFFFFFFFFFFFFFFFF	ULONG_MAX	–1	–1

Unsuffixed constants can lead to type ambiguities that can affect other parts of your program, such as when the results of **sizeof** operations are assigned to variables.. For example, in 32-bit mode, the compiler types a number like 4294967295 (**UINT_MAX**) as an unsigned long and **sizeof** returns 4 bytes. In 64-bit mode, this same number becomes a signed long and **sizeof** will return 8 bytes. Similar problems occur when passing constants directly to functions.

You can avoid these problems by using the suffixes **L** (for long constants) or **UL** (for unsigned long constants) to explicitly type all constants that have the potential of affecting assignment or expression evaluation in other parts of your program. In the example cited above, suffixing the number as `4294967295U` forces the compiler to always recognize the constant as an **unsigned int** in 32-bit or 64-bit mode.

Bit-shifting long values

Left bit-shifting long values will produce different results in 32-bit and 64-bit modes. The examples in the table below show the effects of performing a bit-shift on long constants, using the following code segment:

```
long l=valueL<<1;
```

Table 4. Results of bit-shifting long values

Initial value	Symbolic constant	Value after bit shift	
		32-bit mode	64-bit mode
0x7FFFFFFFL	INT_MAX	0xFFFFFFFE	0x00000000FFFFFFFE
0x80000000L	INT_MIN	0x00000000	0x0000000100000000
0xFFFFFFFFL	UINT_MAX	0xFFFFFFFE	0x1FFFFFFFE

Assigning pointers

In 64-bit mode, pointers and **int** types are no longer the same size. The implications of this are:

- Exchanging pointers and **int** types causes segmentation faults.
- Passing pointers to a function expecting an **int** type results in truncation.
- Functions that return a pointer, but are not explicitly prototyped as such, return an **int** instead and truncate the resulting pointer, as illustrated in the following example.

Although code constructs such as the following are valid in 32-bit mode:

```
a=(char*) calloc(25);
```

without a function prototype for **calloc**, the compiler assumes the function returns an **int**, so a is silently truncated, and then sign-extended. Type casting the result will not prevent the truncation, as the address of the memory allocated by **calloc** was already truncated during the return. In this example, the correct solution would be to include the appropriate header file, **stdlib.h**, which contains the prototype for **calloc**.

To avoid these types of problems:

- Prototype any functions that return a pointer.
- Be sure that the type of parameter you are passing in a function (pointer or **int**) call matches the type expected by the function being called.
- For applications that treat pointers as an integer type, use type **long** or **unsigned long** in either 32-bit or 64-bit mode.

Aligning aggregate data

Structures are aligned according to the strictest aligned member in both 32-bit and 64-bit modes. However, since long types and pointers change size and alignment in 64-bit, the alignment of a structure's strictest member can change, resulting in changes to the alignment of the structure itself.

Structures that contain pointers or long types cannot be shared between 32-bit and 64-bit applications. Unions that attempt to share **long** and **int** types, or overlay pointers onto **int** types can change or corrupt the alignment. In general, you should check all but the simplest structures for alignment and size dependencies.

For detailed information on aligning data structures, including structures that contain bit fields, see Chapter 2, "Aligning data in aggregates," on page 5.

Calling Fortran code

A significant number of applications use C, C++, and Fortran together, by calling each other or sharing files. It is currently easier to modify data sizes and types on the C side than the on Fortran side of such applications. The following table lists C and C++ types and the equivalent Fortran types in the different modes.

Table 5. Equivalent C/C++ and Fortran data types

C/C++ type	Fortran type	
	32-bit	64-bit
signed int	INTEGER	INTEGER
signed long	INTEGER	INTEGER*8
unsigned long	LOGICAL	LOGICAL*8
pointer	INTEGER	INTEGER*8
		POINTER (4 bytes)
		POINTER*8 (8 bytes)

Chapter 2. Aligning data in aggregates

XL C/C++ provides many mechanisms for specifying data alignment at the levels of individual variables, members of aggregates, entire aggregates, and entire compilation units. If you are porting applications between different platforms, or between 32-bit and 64-bit modes, you will need to take into account the differences between alignment settings available in the different environments, to prevent possible data corruption and deterioration in performance.

"Using alignment modes and modifiers" discusses the default alignment settings for all data types on the different platforms and addressing models; options you can use to control the alignment of aggregates and aggregate members; and general rules for aggregate alignment.

"Using and aligning bit fields" on page 8 discusses additional rules and considerations for the use and alignment of bit fields, and provides an example of bit-packed alignment.

Using alignment modes and modifiers

Within aggregates that contain different data types, including C and C++ structures and unions, and C++ classes, each data type supported by XL C/C++ is aligned along byte boundaries according to platform-specific defaults, as follows:

- AIX **power** or **full**, which are equivalent.
- Linux **linuxppc**.
- Mac OS X **power**.

Each of these settings is defined in Table 6 on page 6.

You can also explicitly control the alignment of data by using an alignment *mode*, as well as alignment *modifiers*. Alignment *modes* allow you to do the following:

Set the alignment for all aggregates in a single file or multiple files in the compilation process
To use this approach, you specify the **-qalign** compiler option during compilation. The valid suboptions for **-qalign** for each platform are provided in Table 6 on page 6.

Set the alignment for a single aggregate or multiple aggregates in a file
To use this approach, you specify the **#pragma align** or **#pragma options align** directives in the source files. The valid suboptions for **#pragma align** for each platform are provided in Table 6 on page 6. Each directive changes the alignment rule in effect for all aggregates that follow the directive until another directive is encountered, or until the end of the compilation unit.

Set the alignment for a single aggregate
In addition to the **#pragma align** directive, you can use the following in source files:

- Include the **__attribute__((aligned(n)))** type attribute in structure declarations. The value for *n* must be a positive power of 2. For the correct syntax for using **__attribute__((aligned))** as a type attribute for an aggregate, see "Type Attributes" in *XL C/C++ Language Reference*.

- Include the **__align(n)** specifier in structure declarations. The value for *n* must be a positive power of 2.

Alignment *modifiers* allow you to do the following:

Set the alignment for all members in an aggregate
To use this approach, you can use any of the following in source files:

- Include the **#pragma pack** directive before structure declarations. For valid values for this directive, see **#pragma pack** in *XL C/C++ Compiler Reference*.
- Include the **__attribute__((packed))** type attribute in structure declarations. For the correct syntax for using **__attribute__((packed))** as a type attribute, see "Type Attributes" in *XL C/C++ Language Reference*.

Set the alignment for a single member within an aggregate
To use this approach, include **__attribute__((packed))** or **__attribute__((aligned(n)))** type or variable attributes in structure declarations. The value for *n* in **__attribute__((aligned))** must be a positive power of 2. For more information on the variable attributes, see "The aligned Variable Attribute" and "The packed Variable Attribute" in *XL C/C++ Language Reference*. For information on the type attributes, see "Type Attributes" in *XL C/C++ Language Reference*.

Note: The **__align** specifier and **__attribute__((aligned))** attribute do not change the alignment of vector types.

Table 6. Alignment settings

		Alignment settings and supported platforms							
		natural	**power**	**full**	**mac68k**	**twobyte**	**linuxppc**	**bit_packed**	**packed**
Data type	**Storage**	**AIX Mac**	**AIX Mac**	**AIX**	**AIX Mac**	**AIX**	**Linux**	**AIX Mac Linux**	**AIX**
_Bool (C), bool (C++), _Bool	1 byte	n/a	n/a		n/a		1 byte	1 byte	n/a
char, signed char, unsigned char	1 byte	1 byte	1 byte		1 byte		1 byte	1 byte	
wchar_t (32-bit mode)	2 bytes	2 bytes	2 bytes		2 bytes		2 bytes	1 byte	
wchar_t (64-bit mode)	4 bytes	4 bytes	4 bytes		not supported[2]		4 bytes	1 byte	
int, unsigned int	4 bytes	4 bytes	4 bytes		2 bytes		4 bytes	1 byte	
short int, unsigned short int	2 bytes	2 bytes	2 bytes		2 bytes		2 bytes	1 byte	
long int, unsigned long int (32-bit mode)	4 bytes	4 bytes	4 bytes		2 bytes		4 bytes	1 byte	
long int, unsigned long int (64-bit mode)	8 bytes	8 bytes	8 bytes		not supported[2]		8 bytes	1 byte	
long long	8 bytes	8 bytes	8 bytes		2 bytes		8 bytes	1 byte	
float	4 bytes	4 bytes	4 bytes		2 bytes		4 bytes	1 byte	
double	8 bytes	8 bytes	see note[1]		2 bytes		8 bytes	1 byte	
long double	8 bytes	8 bytes	see note[1]		2 bytes		8 bytes	1 byte	
pointer (32-bit mode)	4 bytes	4 bytes	4 bytes		2 bytes		4 bytes	1 byte	
pointer (64-bit mode)	8 bytes	8 bytes	8 bytes		not supported[2]		8 bytes	1 byte	

Table 6. Alignment settings (continued)

Data type	Storage	Alignment settings and supported platforms							
		natural	**power**	**full**	**mac68k**	**twobyte**	**linuxppc**	**bit_packed**	**packed**
		AIX Mac	**AIX Mac**	**AIX**	**AIX Mac**	**AIX**	**Linux**	**AIX Mac Linux**	**AIX**
vector types[3]	16 bytes	16 bytes	16 bytes	n/a	16 bytes	n/a	16 bytes	1 byte	n/a

Notes:

1. These types use the **natural** alignment for the first member in the aggregate and 4 bytes or the **natural** alignment (whichever is less) for subsequent members.
2. If you declare an aggregate with a member of this type and try to compile with this alignment setting, the compiler issues a warning message, and compiles with the default alignment setting for the appropriate platform.
3. Only supported when the **-qaltivec** compiler option is in effect.

If you generate data with an application on one platform and read the data with an application on another platform, you will want to be sure to use a platform-neutral alignment mode, such as **#pragma pack** or **qalign=bit_packed**.

Note: C++ The C++ compiler might generate extra fields for classes that contain base classes or virtual functions. Objects of these types might not conform to the usual mappings for aggregates.

General rules for alignment

If you control the alignment of aggregates with any of the settings listed in Table 6 on page 6, the following rules apply:

- For all alignment settings, the *size* of an aggregate is the smallest multiple of its alignment value that can encompass all of the members of the aggregate.
- For all alignment settings except **mac68k**, the *alignment* of an aggregate is equal to the largest alignment value of any of its members.
- Aligned aggregates can be nested, and the alignment rules applicable to each nested aggregate are determined by the alignment mode that is in effect when a nested aggregate is declared.

For rules on aligning aggregates containing bit fields, see “Using and aligning bit fields” on page 8.

Related references

- **-qalign** in *XL C/C++ Compiler Reference*
- **#pragma align** in *XL C/C++ Compiler Reference*
- **#pragma pack** in *XL C/C++ Compiler Reference*
- "The **aligned** Variable Attribute", "The **packed** Variable Attribute", "The **__align** Specifier", and "Type Attributes" in "Declarations" in *XL C/C++ Language Reference*.
- **-qaltivec** in *XL C/C++ Compiler Reference*

Using and aligning bit fields

You can declare a bit field as a **_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. A bit field is always 4 or 8 bytes, depending on the declared base type and the compilation mode (32-bit or 64-bit).

C In the C language, you can specify bit fields as **char** or **short** instead of **int**, but XL C/C++ maps them as if they were **unsigned int**. The length of a bit field cannot exceed the length of its base type. In extended mode, you can use the **sizeof** operator on a bit field. (The **sizeof** operator on a bit field always returns 4.)

C++ The length of a bit field can exceed the length of its base type, but the remaining bits will be used to pad the field, and will not actually store any value.

However, alignment rules for aggregates containing bit fields are different depending on the alignment setting you specify. These rules are described below.

Rules for Linux PowerPC alignment

- Bit fields are allocated from a bit field container. The size of this container is determined by the declared type of the bit field. For example, a **char** bit field uses an 8-bit container, an **int** bit field uses 32 bits, and so on. The container must be large enough to contain the bit field, as the bit field will not be split across containers.
- Containers are aligned in the aggregate as if they start on a natural boundary for that type of container. Bit fields are not necessarily allocated at the start of the container.
- If a zero-length bit field is the first member of an aggregate, it has no effect on the alignment of the aggregate and is overlapped by the next data member. If a zero-length bit field is a non-first member of the aggregate, it pads to the next alignment boundary determined by its base declared type but does not affect the alignment of the aggregate.
- Unnamed bit fields do not affect the alignment of the aggregate.

Rules for bit-packed alignment

- Bit fields have an alignment of 1 byte, and are packed with no default padding between bit fields.
- A zero-length bit field causes the next member to start at the next byte boundary. If the zero-length bit field is already at a byte boundary, the next member starts at this boundary. A non-bit field member that follows a bit field is aligned on the next byte boundary.

Example of bit field alignment

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
```

The size of A is 7 bytes. The alignment of A is 1 byte. The layout of A is:

Member name	Byte offset	Bit offset
a	0	0
b	1	0
c	2	2
d	3	6
e	4	2
f	5	0
g	6	0

Chapter 3. Handling floating point operations

XL C/C++ supports single-precision floating-point numbers with an approximate range of 10^{-38} to 10^{+38}, and about 7 decimal digits of precision; and double-precision floating-point numbers with an approximate range of 10^{-308} to 10^{+308} and precision of about 16 decimal digits.

The following sections provide reference information, portability considerations, and suggested procedures for using compiler options to manage floating-point operations:

Handling multiply-add operations

By default, the compiler violates certain IEEE 754 floating-point rules in order to improve performance. For example, multiply-add instructions are generated by default because they are faster and produce a more precise result than separate multiply and add instructions. If you want greater compatibility with the accuracy available on other systems, you can use the **-qfloat=nomaf** option to suppress the generation of these multiply-add instructions.

Related references

- **-qfloat** in *XL C/C++ Compiler Reference*

Handling floating-point rounding

By default, the compiler attempts to perform as much arithmetic as possible at compile time. A floating-point operation with constant operands is *folded*, which means that the arithmetical expression is replaced with the compile-time result. If you enable optimization, increased folding might occur. However, the result of a compile-time computation might differ slightly from the result that would have been calculated at run time, because more rounding operations occur at compile time. For example, where a multiply-add-fused (MAF) operation might be used at run time with less rounding, separate multiply and add operations might be used at compile time, producing a slightly different result.

To prevent the possibility of unexpected results due to compile-time rounding, you have two options:

- Use the **-qfloat=nofold** compiler option to suppress all compile-time folding of floating-point computations.
- Use the **-y** compiler option to specify a IEEE compile-time rounding mode that matches the rounding mode to be used at run time. By default, the rounding mode is *round-to-nearest*, unless you specify another value (which you can do via the XL C/C++ built-in function **__setrnd**, declared in the **builtins.h** file).

For example, if you were to compile the following code sample with **-yz**, which specifies a rounding mode of round-to-zero, the two results of `u.x` would be slightly different:

```
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;
 }
```

This is because the calculation of `1.0/3.0` would be folded at compile-time using round-to-zero rounding, while `one/three` would be calculated at run-time using the default rounding mode of round-to-nearest. (Declaring the variables `one` and `three` as **volatile** suppresses folding by the compiler, even under optimization.) The output of the program would be:

```
1/3=3EAAAAAA
1/3=3EAAAAAB
```

To ensure consistency between compile-time and run-time results in this example, you would compile with the option **-yn** (which is the default).

Related references

- **-qfloat** in *XL C/C++ Compiler Reference*
- **-y** in *XL C/C++ Compiler Reference*
- **__setrnd** in "Appendix B: Built-in Functions" in the*XL C/C++ Compiler Reference*

Handling floating-point exceptions

By default, invalid operations such as division by zero, division by infinity, overflow, and underflow are ignored at run time. However, you can use the **-qflttrap** option to detect these types of exceptions. In addition, you can add suitable support code to your program to allow program execution to continue after an exception occurs, and to modify the results of operations causing exceptions.

Because, however, floating-point computations involving constants are usually folded at compile time, the potential exceptions that would be produced at run time will not occur. To ensure that the **-qflttrap** option traps all run-time floating-point exceptions, consider using the **-qfloat=nofold** option to suppress all compile-time folding.

Related references

- **-qfloat** in *XL C/C++ Compiler Reference*
- **-qflttrap** in *XL C/C++ Compiler Reference*

Using the Mathematical Acceleration Subsystem (MASS)

The XL C/C++ Advanced Edition V7.0 for Linux ships the Mathematical Acceleration Subsystem (MASS), a set of libraries of tuned mathematical intrinsic functions that provide improved performance over the corresponding **libm.a** library functions. The accuracy and exception handling might not be identical in MASS functions and **libm.a** functions.

The MASS libraries on Linux consist of a library of vector functions, described in "Using the vector libraries." "Compiling and linking a program with MASS" describes how to compile and link a program that uses the MASS libraries.

Using the vector libraries

The MASS libraries for Linux are a subset of the MASS libraries for AIX. On Linux, 32-bit and 64-bit objects must not be mixed in a single library, so two versions of the MASS library are provided: **libmassvp4.a** (32-bit) and **libmassvp4_64.a** (64-bit).

The single-precision and double-precision functions contained in the vector libraries are summarized in Table 7. To provide the prototypes for the functions, include **massv.h** in your source files. Note that in C and C++ applications, only call by reference is supported, even for scalar arguments.

Table 7. MASS vector library functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vrec	vsrec	Sets y[i] to the reciprocal of x[i], for i=0,..,*n–1	void vrec (double y[], double x[], int *n);	void vsrec (float y[], float x[], int *n);
vrsqrt	vsrsqrt	Sets y[i] to the reciprocal of the square root of x[i], for i=0,..,*n–1	void vrsqrt (double y[], double x[], int *n);	void vsrsqrt (float y[], float x[], int *n);
vsqrt	vssqrt	Sets y[i] to the square root of x[i], for i=0,..,*n–1	void vsqrt (double y[], double x[], int *n);	void vssqrt (float y[], float x[], int *n);

Consistency of MASS vector functions

All the functions in the MASS vector libraries are consistent, in the sense that a given input value will always produce the same result, regardless of its position in the vector, and regardless of the vector length.

Compiling and linking a program with MASS

To compile an application that calls the routines in the MASS libraries, specify **massvp4** (32-bit) or **massvp4_64** (64-bit) on the **-l** linker option. For example, if the MASS libraries are installed in the default directory, you could specify one of the following:

```
xlc progc.c -o progf -lmassvp4
xlc progc.c -o progf -lmassvp4_64 -q64
```

The MASS functions must run in the round-to-nearest rounding mode and with floating-point exception trapping disabled. (These are the default compilation settings.)

Chapter 4. Using C++ templates

In C++, you can use a template to declare a set of related:

- Classes (including structures)
- Functions
- Static data members of template classes

Within an application, you can instantiate the same template multiple times with the same arguments or with different arguments. If you use the same arguments, the repeated instantiations are redundant. These redundant instantiations increase compilation time, increase the size of the executable, and deliver no benefit.

There are four basic approaches to the problem of redundant instantiations:

Code for unique instantiations
Organize your source code so that the object files contain only one instance of each required instantiation and no unused instantiations. This is the least usable approach, because you must know where each template is defined and where each template instantiation is required.

Instantiate at every occurrence
Use the **-qnotempinc** and **-qnotemplateregistry** compiler options (these are the default settings). The compiler generates code for every instantiation that it encounters. With this approach, you accept the disadvantages of redundant instantiations.

Have the compiler store instantiations in a template include directory
Use the **-qtempinc** compiler option. If the template definition and implementation files have the required structure, each template instantiation is stored in a template include directory. If the compiler is asked to instantiate the same template again with the same arguments, it uses the stored version instead. This approach is described in "Using the -qtempinc compiler option."

Have the compiler store instantiation information in a registry
Use the **-qtemplateregistry** compiler option. Information about each template instantiation is stored in a template registry. If the compiler is asked to instantiate the same template again with the same arguments, it points to the instantiation in the first object file instead. The **-qtemplateregistry** compiler option provides the benefits of the **-qtempinc** compiler option but does not require a specific structure for the template definition and implementation files. This approach is described in "Using the -qtemplateregistry compiler option" on page 18.

Note: The **-qtempinc** and **-qtemplateregistry** compiler options are mutually exclusive.

Using the -qtempinc compiler option

To use **-qtempinc**, you must structure your application as follows:

- Declare your class templates and function templates in template header files, with a **.h** extension.

- For each template declaration file, create a template implementation file. This file must have the same file name as the template declaration file and an extension of **.c** or **.t**, or the name must be specified in a **#pragma implementation** directive. For a class template, the implementation file defines the member functions and static data members. For a function template, the implementation file defines the function.
- In your source program, specify an **#include** directive for each template declaration file.
- Optionally, to ensure that your code is applicable for both **-qtempinc** and **-qnotempinc** compilations, in each template declaration file, conditionally include the corresponding template implementation file if the **__TEMPINC__** macro is *not* defined. (This macro is automatically defined when you use the **-qtempinc** compilation option.)

  This produces the following results:

  – Whenever you compile with **-qnotempinc**, the template implementation file is included.

  – Whenever you compile with **-qtempinc**, the compiler does not include the template implementation file. Instead, the compiler looks for a file with the same name as the template implementation file and extension **.c** the first time it needs a particular instantiation. If the compiler subsequently needs the same instantiation, it uses the copy stored in the template include directory.

Example of -qtempinc

This example includes the following source files:

- A template declaration file: `stack.h`.
- The corresponding template implementation file: `stack.c`.
- A function prototype: `stackops.h` (not a function template).
- The corresponding function implementation file: `stackops.cpp`.
- The main program source file: `stackadd.cpp`.

In this example:

1. Both source files include the template declaration file `stack.h`.
2. Both source files include the function prototype `stackops.h`.
3. The template declaration file conditionally includes the template implementation file `stack.c` if the program is compiled with **-qnotempinc**.

Template declaration file: stack.h

This header file defines the class template for the class `Stack`.

```
#ifndef STACK_H
#define STACK_H

template <class Item, int size> class Stack {
  public:
      void push(Item item);  // Push operator
      Item pop();            // Pop operator
      int isEmpty(){
          return (top==0);   // Returns true if empty, otherwise false
      }
      Stack() { top = 0; }   // Constructor defined inline
  private:
      Item stack[size];      // The stack of items
      int   top;             // Index to top of stack
};
```

```
#ifndef __TEMPINC__           //  3
#include "stack.c"            //  3
#endif                        //  3
#endif
```

Template implementation file: stack.c

This file provides the implementation of the class template for the class `Stack`.

```
template <class Item, int size>
    void Stack<Item,size>::push(Item item) {
    if (top >= size) throw size;
    stack[top++] = item;
      }
template <class Item, int size>
    Item Stack<Item,size>::pop() {
    if (top <= 0) throw size;
    Item item = stack[--top];
    return(item);
      }
```

Function declaration file: stackops.h

This header file contains the prototype for the `add` function, which is used in both `stackadd.cpp` and `stackops.cpp`.

```
void add(Stack<int, 50>& s);
```

Function implementation file: stackops.cpp

This file provides the implementation of the `add` function, which is called from the main program.

```
#include "stack.h"            //  1
#include "stackops.h"         //  2

void add(Stack<int, 50>& s) {
  int tot = s.pop() + s.pop();
  s.push(tot);
  return;
    }
```

Main program file: stackadd.cpp

This file creates a `Stack` object.

```
#include <iostream.h>
#include "stack.h"            //  1
#include "stackops.h"         //  2

main() {
  Stack<int, 50> s;           // create a stack of ints
  int left=10, right=20;
  int sum;

  s.push(left);               // push 10 on the stack
  s.push(right);              // push 20 on the stack
  add(s);                     // pop the 2 numbers off the stack
                                  // and push the sum onto the stack
  sum = s.pop();              // pop the sum off the stack

  cout << "The sum of: " << left << " and: " << right << " is: " << sum << endl;

  return(0);
    }
```

Regenerating the template instantiation file

The compiler builds a template instantiation file in the **TEMPINC** directory corresponding to each template implementation file. With each compilation, the compiler can add information to the file but it never removes information from the file.

As you develop your program, you might remove template function references or reorganize your program so that the template instantiation files become obsolete. You can periodically delete the **TEMPINC** destination and recompile your program.

Using -qtempinc with shared libraries

In a traditional application development environment, different applications can share both source files and compiled files. When you use templates, applications can share source files but cannot share compiled files.

If you use **-qtempinc**:

- Each application must have its own **TEMPINC** destination.
- You must compile all of the source files for the application, even if some of the files have already been compiled for another application.

Related references

- **-qtempinc** in *XL C/C++ Compiler Reference*
- **#pragma implementation** in *XL C/C++ Compiler Reference*

Using the -qtemplateregistry compiler option

Unlike **-qtempinc**, the **-qtemplateregistry** compiler option does not impose specific requirements on the organization of your source code. Any program that compiles successfully with **-qnotempinc** will compile with **-qtemplateregistry**.

The template registry uses a ″first-come first-served″ algorithm:

- When a program references a new instantiation for the first time, it is instantiated in the compilation unit in which it occurs.
- When another compilation unit references the same instantiation, it is not instantiated. Thus, only one copy is generated for the entire program.

The instantiation information is stored in a template registry file. You must use the same template registry file for the entire program. Two programs cannot share a template registry file.

The default file name for the template registry file is **templateregistry**, but you can specify any other valid file name to override this default. When cleaning your program build environment before starting a fresh or scratch build, you must delete the registry file along with the old object files.

Recompiling related compilation units

If two compilation units, A and B, reference the same instantiation, the **-qtemplateregistry** compiler option has the following effect:

- If you compile A first, the object file for A contains the code for the instantiation.
- When you later compile B, the object file for B does not contain the code for the instantiation because object A already does.

- If you later change A so that it no longer references this instantiation, the reference in object B would produce an unresolved symbol error. When you recompile A, the compiler detects this problem and handles it as follows:
  - If the **-qtemplaterecompile** compiler option is in effect, the compiler automatically recompiles B during the link step, using the same compiler options that were specified for A. (Note, however, that if you use separate compilation and linkage steps, you need to include the compilation options in the link step to ensure the correct compilation of B.)
  - If the **-qnotemplaterecompile** compiler option is in effect, the compiler issues a warning and you must manually recompile B.

Switching from -qtempinc to -qtemplateregistry

Because the **-qtemplateregistry** compiler option does not impose any restrictions on the file structure of your application, it has less administrative overhead than **-qtempinc**. You can make the switch as follows:

- If your application compiles successfully with both **-qtempinc** and **-qnotempinc**, you do not need to make any changes.
- If your application compiles successfully with **-qtempinc** but not with **-qnotempinc**, you must change it so that it will compile successfully with **-qnotempinc**. In each template definition file, conditionally include the corresponding template implementation file if the **__TEMPINC__** macro is not defined. This is illustrated in "Example of -qtempinc" on page 16.

Related references

- **-qtemplateregistry** in *XL C/C++ Compiler Reference*
- **-qtemplaterecompile** in *XL C/C++ Compiler Reference*

Chapter 5. Constructing a library

You can include static and shared libraries in your C and C++ applications.

"Compiling and linking a library" describes how to compile your source files into object files for inclusion in a library, how to link a library into the main program, and how to link one library into another.

"Initializing static objects in libraries (C++)" on page 22 describes how to use *priorities* to control the order of initialization of objects across multiple files in a C++ application.

Compiling and linking a library

Compiling a static library

To compile a static library:

1. Compile each source file into an object file, with no linking.
2. Use the GCC **ar** command to add the generated object files to an archive library file.

For example:

```
xlc -c bar.c example.c
ar -rv libfoo.a bar.o example.o
```

Compiling a shared library

To compile a shared library:

1. Compile your source files into an object file, with no linking. For example:

   ```
   xlc -c foo.c
   ```
2. Use the **-qmkshrobj** compiler option to create a shared object from the generated object files. For example:

   ```
   xlc -qmkshrobj -o libfoo.so foo.o
   ```

Linking a library to an application

You can use the same command string to link a static or shared library to your main program. For example:

```
xlc -o myprogram main.c -Ldirectory -lfoo
```

where *directory* is the path to the directory containing the library.

By using the **-l** option, you instruct the linker to search in the directory specified via the **-L** option for `libfoo.so`; if it is not found, the linker searches for `libfoo.a`. For additional linkage options, including options that modify the default behavior, see the GCC **ld** documentation.

Linking a shared library to another shared library

Just as you link modules into an application, you can create dependencies between shared libraries by linking them together. For example:

```
xlc -qmkshrobj -o mylib.so myfile.o -Ldirectory -lfoo
```

Related references

- **-qmkshrobj** in *XL C/C++ Compiler Reference*
- **-l** in *XL C/C++ Compiler Reference*
- **-L** in *XL C/C++ Compiler Reference*

Initializing static objects in libraries (C++)

The C++ language definition specifies that, before the **main** function in a C++ program is executed, all objects with constructors, from all the files included in the program must be properly constructed. Although the language definition specifies the order of initialization for these objects *within* a file (which follows the order in which they are declared), it does not, however, specify the order of initialization for these objects *across* files and libraries. You might want to specify the initialization order of static objects declared in various files and libraries in your program.

To specify an initialization order for objects, you assign relative *priority* numbers to objects. The mechanisms by which you can specify priorities for entire files or objects within files are discussed in "Assigning priorities to objects." The mechanisms by which you can control the initialization order of objects across modules are discussed in "Order of object initialization across libraries" on page 24.

Assigning priorities to objects

You can assign a priority number to objects and files within a single library, and the objects will be initialized at run time according to the order of priority. However, because of the differences in the way modules are loaded and objects initialized on the different platforms, the levels at which you can assign priorities vary among the different platforms, as follows:

AIX Linux **Set the priority level for an entire file**

To use this approach, you specify the **-qpriority** compiler option during compilation. By default, all objects within a single file are assigned the same priority level, and are initialized in the order in which they are declared, and terminated in reverse declaration order.

AIX Linux Mac OS X **Set the priority level for objects within a file**

To use this approach, you include **#pragma priority** directives in the source files. Each **#pragma priority** directive sets the priority level for all objects that follow it, until another pragma directive is specified. Within a file, the first **#pragma priority** directive must have a higher priority number than the number specified in the **-qpriority** option (if it is used), and subsequent **#pragma priority** directives must have increasing numbers. While the relative priority of objects *within* a single file will remain the order in which they are declared, the pragma directives will affect the order in which objects are initialized *across* files. The objects are initialized according to their priority, and terminated in reverse priority order.

Linux Mac OS X **Set the priority level for individual objects**

To use this approach, you use **init_priority** variable attributes in the source files. The **init_priority** attribute takes precedence over **#pragma priority** directives, and can be applied to objects in any declaration order. On Linux, the objects are initialized according to their priority and terminated

in reverse priority *across* compilation units; on Mac OS X, the objects are initialized according to their priority and terminated in reverse priority only *within* a compilation unit.

AIX On AIX only, you can additionally set the priority of an entire shared library, by using the priority sub-option of the **-qmkshrobj** compiler option. As loading and initialization on AIX occur as separate processes, priority numbers assigned to files (or to objects within files) are entirely independent of priority numbers assigned to libraries, and do not need to follow any sequence.

Using priority numbers

AIX Priority numbers can range from -2147483643 to 2147483647. However, numbers from -2147483648 to -2147482624 are reserved for system use. The smallest priority number that you can specify, -2147482623, is initialized first. The largest priority number, 2147483647, is initialized last. If you do not specify a priority level, the default priority is 0 (zero).

Linux Mac OS X 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. If you do not specify a priority level, the default priority is 65535.

The examples below show how to specify the priority of objects within a single file, and across two files. "Order of object initialization across libraries" on page 24 provides detailed information on the order of initialization of objects on the Linux platform.

Example of object initialization within a file

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 Base a ;

House b ;
...
#pragma priority(3000) //Following objects constructed with priority 3000
...

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

Garage d ;
...
```

Example of object initialization across multiple files

The following example describes the initialization order for objects in two files, `farm.C` and `zoo.C`. Both files use **#pragma priority** directives and are compiled with the **-qpriority** option.

farm.C -qpriority=2000

```
#pragma priority(3000)
...
Dog a ;
Dog b ;
...
#pragma priority(6000)
...
Cat c ;
Cow d ;
...
#pragma priority(7000)
Mouse e ;
...
```

zoo.C -qpriority=2000

```
...
Lion k ;
#pragma priority(4000)
Bear m ;
...
#pragma priority(5000)
...
Zebra n ;
Snake s ;
...
#pragma priority(8000)
Frog f ;
...
```

At run time, the objects in these files are initialized in the following order:

Sequence	Object	Priority value	Comment
1	Lion k	2000	Takes priority number of file `zoo.o` (2000) (initialized first).
2	Dog a	3000	Takes pragma priority (3000).
3	Dog b	3000	Follows `Dog a`.
4	Bear m	4000	Next priority number, specified by pragma (4000).
5	Zebra n	5000	Next priority number from pragma (5000).
6	Snake s	5000	Follows with same priority.
7	Cat c	6000	Next priority number.
8	Cow d	6000	Follows with same priority.
9	Mouse e	7000	Next priority number.
10	Frog f	8000	Next priority number (initialized last).

Related references

- **-qpriority** in *XL C/C++ Compiler Reference*
- **#pragma priority** in *XL C/C++ Compiler Reference*
- "The **init_priority** Variable Attribute" in "Declarations" in *XL C/C++ Language Reference*.

Order of object initialization across libraries

At run time, once all modules in an application have been loaded, the modules are initialized in their order of priority (the executable program containing the **main** function is always assigned a priority of 0). When objects are initialized within a library, the order of initialization follows the rules outlined in "Assigning priorities to objects" on page 22. If objects do not have priorities assigned, or have the same priorities, object files are initialized in random order, and the objects within the files are initialized according to their declaration order. Objects are terminated in reverse order of their construction.

Each static library and shared library is loaded and initialized at run time in *reverse* link order, once all of its dependencies have been loaded and initialized. Link order is the order in which each library was listed on the command line during linking into the main application. For example, if library A calls library B, library B is loaded before library A.

As each module is loaded, objects are initialized in order of priority, according to the rules outlined in “Assigning priorities to objects” on page 22. If objects do not have priorities assigned, or have the same priorities, object files are initialized in reverse link order — where link order is the order in which the files were given on the command line during linking into the library — and the objects within the files are initialized according to their declaration order. Objects are terminated in reverse order of their construction.

Example of object initialization across libraries

In this example, the following modules are used:

- `main.out`, the executable containing the main function
- `libS1` and `libS2`, two shared libraries
- `libS3` and `libS4`, two shared libraries that are dependencies of `libS1`
- `libS5` and `libS6`, two shared libraries that are dependencies of `libS2`

The dependent libraries are created with the following command strings:

```
xlC -qmkshrobj -o libS3 fileE.o fileF.o
xlC -qmkshrobj -o libS4 fileG.o fileH.o
xlC -qmkshrobj -o libS5 fileI.o fileJ.o
xlC -qmkshrobj -o libS6 fileK.o fileL.o
```

The dependent libraries are linked with their parent libraries using the following command strings:

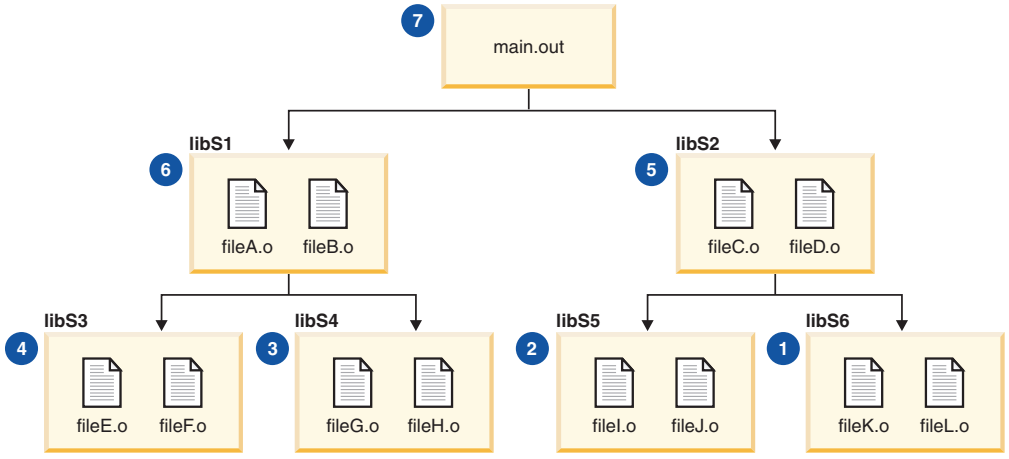
```
xlC -qmkshrobj libS1 fileA.o fileB.o -L. -lS3 -lS4
xlC -qmkshrobj libS2 fileC.o fileD.o -L. -lS5 -lS6
```

The parent libraries are linked with the main program with the following command string:

```
xlC main.c -o main.out -L. -lS1 -lS2
```

The following diagram shows the initialization order of the shared libraries.

Figure 1. Object initialization order on Linux



Objects are initialized as follows:

Sequence	Object	Comment
1	libS6	libS2 was entered last on the command line when linked with main, and so is initialized before libS1. However, libS5 and libS6 are dependencies of libS2, so they are initialized first. Since it was entered last on the command line when linked with libS2, libS6 is initialized first. The objects in this library are initialized according to their priority. (If no priorities are assigned, the objects in fileL are initialized before those in fileK, because fileL was listed last on the command line when the object files were linked into libS6.)
2	libS5	libS5 was entered before libS6 on the command line when linked with libS2, so it is initialized next. The objects in this library are initialized according to their priority. (If no priorities are assigned, the objects in fileJ are initialized before those in fileI, because fileJ was listed last on the command line when the object files were linked into libS5.)
3	libS4	libS4 is a dependency of libS1 and was entered last on the command line during the linking with libS1, so it is initialized next. The objects in this library are initialized according to their priority. (If no priorities are assigned, the objects in fileH are initialized before those in fileG, because fileH was listed last on the command line when the object files were linked into libS4.)
4	libS3	libS3 is a dependency of libS1 and was entered first on the command line during the linking with libS1, so it is initialized next. The objects in this library are initialized according to their priority. (If no priorities are assigned, the objects in fileF are initialized before those in fileE, because fileF was listed last on the command line when the object files were linked into libS3.)
5	libS2	libS2 is initialized next. The objects in this library are initialized according to their priority. (If no priorities are assigned, the objects in fileD are initialized before those in fileC, because fileD was listed last on the command line when the object files were linked into libS2.)
6	libS1	libS1 is initialized next. The objects in this library are initialized according to their priority. (If no priorities are assigned, the objects in fileB are initialized before those in fileA, because fileB was listed last on the command line when the object files were linked into libS1.)
7	main.out	Initialized last. The objects in main.out are initialized according to their priority.

Chapter 6. Optimizing your applications

By default, a standard compilation performs only very basic local optimizations on your code, while still providing fast compilation and full debugging support. Once you have developed, tested, and debugged your code, you will want to take advantage of the extensive range of optimization capabilities offered by XL C/C++, that allow for significant performance gains without the need for any manual re-coding effort. In fact, it is not recommended to excessively hand-optimize your code (for example, by manually unrolling loops), as unusual constructs can confuse the compiler, and make your application difficult to optimize for new machines.

Instead, you can control XL C/C++ compiler optimization through the use of a set of compiler options. These options provide you with the following approaches to optimizing your code:

- You can use an option that performs a specific type of optimization, including:
  - System architecture. If your application will run on a specific hardware configuration, the compiler can generate instructions that are optimized for the target machine, including microprocessor architecture, cache or memory geometry, and addressing model. These options are discussed in "Optimizing for system architecture" on page 31.
  - Shared memory parallelization. If your application will run on hardware that supports shared memory parallelization, you can instruct the compiler to automatically generate threaded code, or to recognize OpenMP standard programming constructs. Options for parallelizing your program are discussed in "Using shared-memory parallelism" on page 33.
  - High-order loop analysis and transformation. The compiler uses various techniques to optimize loops. These options are discussed in "Using high-order loop analysis and transformations" on page 32.
  - Interprocedural analysis (IPA). The compiler reorganizes code sections to optimize calls between functions. IPA options are discussed in "Using interprocedural analysis" on page 34.
  - Profile-directed feedback (PDF). The compiler can optimize sections of your code based on call and block counts and execution times. PDF options are discussed in "Using profile-directed feedback" on page 35
  - Other types of optimization, including loop unrolling, function inlining, stack storage compacting, and many others. Brief descriptions of these options are provided in "Other optimization options" on page 38.
- You can use an optimization *level*, which bundles several techniques and may include one or more of the aforementioned specific optimization options. There are four optimization levels that perform increasingly aggressive optimizations on your code. Optimization levels are described in "Using optimization levels" on page 28.
- You can combine optimization options and levels to achieve the precise results you want. Discussions on how to do so are provided throughout the sections referenced above.

Keep in mind that program optimization implies a trade-off, in that it results in longer compile times, increased program size and disk usage, and diminished debugging capability. At higher levels of optimization, program semantics might be affected, and code that executed correctly before optimization might no longer run as expected. Thus, not all optimizations are beneficial for all applications or even

all portions of applications. For programs that are not computationally intensive, the benefits of faster instruction sequences brought about by optimization can be outweighed by better paging and cache performance brought about by a smaller program footprint.

To identify modules of your code that would benefit from performance enhancements, compile the selected files with the **-p** or **-pg** options, and use the operating system profiler **gprof** to identify functions that are ″hot spots″ and are computationally intensive. If both size and speed are important, optimize the modules which contain hot spots, while keeping code size compact in other modules. To find the right balance, you might need to experiment with different combinations of techniques.

An exhaustive list of all options available for optimization, organized by category, is provided in "Summary of options for optimization and performance" on page 38.

Finally, if you want to manually tune your application to complement the optimization techniques used by the compiler, Chapter 7, "Coding your application to improve performance," on page 41 provides suggestions and best practices for coding for performance.

Related references

- **-p** in *XL C/C++ Compiler Reference*
- **-pg** in *XL C/C++ Compiler Reference*

Using optimization levels

By default, the compiler performs only quick local optimizations such as constant folding and elimination of local common sub-expressions, while still allowing full debugging support. You can optimize your program by specifying various optimization levels, which provide increasing application performance, at the expense of larger program size and debugging support. The options you can specify are summarized in the following table, and more detailed descriptions of the techniques used at each optimization level are provided below.

Table 8. Optimization levels

Option	Behavior
-O or **-O2** or **-qoptimize** or **-qoptimize=2**	Comprehensive low-level optimization; partial debugging support.
-O3 or **-qoptimize=3**	More extensive optimization; some precision trade-offs.
-O4 or **-qoptimize=4**	Interprocedural optimization; loop optimization; automatic machine tuning.
-O5 or **-qoptimize=5**	

Techniques used in optimization level 2

At optimization level 2, the compiler is conservative in the optimization techniques it applies and should not affect program correctness. At optimization level 2, the following techniques are used:

- Eliminating common sub-expressions 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 (equivalent to using the **-qunroll** compiler option).
- Pipelining some loops

Techniques used in optimization level 3

At optimization levels 3 and above, the compiler is more aggressive, making changes to program semantics that will improve performance even if there is some risk that these changes will produce different results. Here are some examples:

- In some cases, X*Y*Z will be calculated as X*(Y*Z) instead of (X*Y)*Z. This could produce a different result due to rounding.
- In some cases, the sign of a negative zero value will be lost. This could produce a different result if you multiply the value by infinity.

"Getting the most out of optimization levels 2 and 3" on page 30 provides some suggestions for mitigating this risk.

At optimization level 3, all of the techniques in optimization level 2 are used, plus the following:

- Unrolling deeper loops and improving loop scheduling.
- Increasing the scope of optimization.
- Performing optimizations with marginal or niche effectiveness, which might not help all programs.
- Performing optimizations that are expensive in compile time or space.
- Reordering some floating-point computations, which might produce precision differences or affect the generation of floating-point-related exceptions (equivalent to compiling with the **-qnostrict** option).
- Eliminating implicit memory usage limits (equivalent to compiling with the **-qmaxmem=-1** option).

- Increasing automatic inlining.
- Propagating constants and values through structure copies.
- Removing the "address taken" attribute if possible after other optimizations.
- Grouping loads, stores and other operations on contiguous aggregate members, in some cases using VMX vector register operations.

Techniques used in optimization levels 4 and 5

At optimization levels 4 and 5, all of the techniques in optimization levels 2 and 3 are used, plus the following:

- Interprocedural analysis, which invokes the optimizer at link time to perform optimizations across multiple source files (equivalent to compiling with the **-qipa** option).
- High-order transformations, which provide optimized handling of loop nests and array language constructs (equivalent to compiling with the **-qhot** option).
- Hardware-specific optimization (equivalent to compiling with the **-qarch=auto**, **-qtune=auto**, and **-qcache=auto** options).
- At optimization level 5, more detailed interprocedural analysis (the equivalent to compiling with the **-qipa=level=2** option). With level 2 IPA, high-order transformations (equivalent to compiling with **-qhot**) are delayed until link time, after whole-program information has been collected.

Getting the most out of optimization levels 2 and 3

Here is a recommended approach to using optimization levels 2 and 3:

1. If possible, test and debug your code without optimization before using **-O2**.
2. Ensure that your code complies with its language standard.
3. C In C code, ensure that the use of pointers follows the type restrictions: generic pointers should be **char*** or **void***. Also check that all shared variables and pointers to shared variables are marked **volatile**.
4. C In C, use the **-qlibansi** compiler option unless your program defines its own functions with the same names as library functions.
5. Compile as much of your code as possible with **-O2**.
6. If you encounter problems with **-O2**, consider using **-qalias=noansi** rather than turning off optimization.
7. Next, use **-O3** on as much code as possible.
8. If you encounter problems or performance degradations, consider using **-qstrict** or **-qcompact** along with **-O3** where necessary.
9. If you still have problems with **-O3**, switch to **-O2** for a subset of files, but consider using **-qmaxmem=-1**, **-qnostrict**, or both.

Related references

- **-O** in *XL C/C++ Compiler Reference*
- **-qnostrict** in *XL C/C++ Compiler Reference*
- **-qmaxmem** in *XL C/C++ Compiler Reference*
- **-qunroll** in *XL C/C++ Compiler Reference*
- **-qalias** in *XL C/C++ Compiler Reference*
- **-qlibansi** in *XL C/C++ Compiler Reference*

Optimizing for system architecture

You can instruct the compiler to generate code for optimal execution on a given microprocessor or architecture family. By selecting appropriate target machine options, you can optimize to suit the broadest possible selection of target processors, a range of processors within a given family of processor architectures, or a specific processor. The following table lists the optimization options that affect individual aspects of the target machine. Using a predefined optimization level sets default values for these individual options.

Table 9. Target machine options

Option	Behavior
-q32	Generates code for a 32-bit (4/4/4) addressing model (32-bit execution mode). This is the default setting.
-q64	Generates code for a 64-bit (4/8/8) addressing model (64-bit execution mode).
-qarch	Selects a family of processor architectures for which instruction code should be generated. This option restricts the instruction set generated to a subset of that for the PowerPC® architecture. The default on all Linux platforms except Y-HPC is **-qarch=ppc64grsq**. The default on Y-HPC is **-qarch=ppc970**. Using **-O4** or **-O5** sets the default to **-qarch=auto**.
-qtune	Biases optimization toward execution on a given microprocessor, without implying anything about the instruction set architecture to use as a target. The default is **-qtune=pwr4**.
-qcache	Defines a specific cache or memory geometry. The defaults are determined through the setting of **-qtune**.
-qenablevmx	Instructs the compiler to generate VMX (AltiVec) code in any compiler phase. For SUSE 9, Y-HPC and Red Hat 4 Linux, this option is enabled by default; for Red Hat 3 Linux, the default is **-qnoenablevmx**.

For a complete listing of valid hardware-related suboptions and combinations of suboptions, see “Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation”, and “Acceptable Compiler Mode and Processor Architecture Combinations” in *XL C/C++ Compiler Reference*.

Getting the most out of target machine options

Using -qarch options

If your application will run on the same machine on which you are compiling it, you can use the **-qarch=auto** option, which automatically detects the specific architecture of the compiling machine, and generates code to take advantage of instructions available only on that machine (or on a system that supports the equivalent processor architecture). Otherwise, try to specify with **-qarch** the smallest family of machines possible that will be expected to run your code reasonably well.

Using -qtune options

If you specify a particular architecture with **-qarch**, **-qtune** will automatically select the suboption that generates instruction sequences with the best performance for that architecture. If you specify a *group* of architectures with **-qarch**, compiling with **-qtune=auto** will generate code that runs on all of the architectures in the specified group, but the instruction sequences will be those with the best performance on the architecture of the compiling machine.

Try to specify with **-qtune** the particular architecture that the compiler should target for best performance but still allow execution of the produced object file on all architectures specified in the**-qarch** option. For information on the valid combinations of **-qarch** and **-qtune**, see "Acceptable Compiler Mode and Processor Architecture Combinations" in *XL C/C++ Compiler Reference*.

Using -qcache options

Before using the **-qcache** option, use the **-qlistopt** option to generate a listing of the current settings and verify if they are satisfactory. If you decide to specify your own **-qcache** suboptions, use **-qhot** or **-qsmp** along with it. For the full set of suboptions, option syntax, and guidelines for use, see **-qcache** in *XL C/C++ Compiler Reference*.

Related references

- **-qarch** in *XL C/C++ Compiler Reference*
- **-qcache** in *XL C/C++ Compiler Reference*
- **-qtune** in *XL C/C++ Compiler Reference*
- **-qenablevmx** in *XL C/C++ Compiler Reference*
- **-qlistopt** in *XL C/C++ Compiler Reference*

Using high-order loop analysis and transformations

High-order transformations are optimizations that specifically improve the performance of loops through techniques such as interchange, fusion, and unrolling. The goals of these loop optimizations include:

- Reducing the costs of memory access through the effective use of caches and translation look-aside buffers.
- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware.
- Improving the utilization of microprocessor resources through reordering and balancing the usage of instructions with complementary resource requirements.

To enable high-order loop analysis and transformations, you use the **-qhot** option. The following table lists the suboptions available for **-qhot**.

Table 10. **-qhot** *suboptions*

suboption	Behavior
vector	Instructs the compiler to transform some loops to use optimized versions of various trigonometric functions and operations such as reciprocal and square root that reside in a built-in library, rather than use the standard versions. The optimized versions make different trade-offs with respect to precision versus performance. This suboption is enabled by default when you use **-qhot**, **-O4**, or **-O5**.
novector	Instructs the compiler to avoid optimizations that use the above-mentioned built-in library functions. Use this suboption or **-qstrict** if you do not want your precision of your program's results to be affected.
arraypad	Instructs the compiler to pad any arrays where it infers there might be a benefit and to pad by whatever amount it chooses.

Table 10. **-qhot** *suboptions (continued)*

suboption	Behavior
simd	Instructs the compiler to attempt automatic SIMD vectorization; that is, converting certain operations in a loop that apply to successive elements of an array into a call to a VMX instruction. This call calculates several results at one time, which is faster than calculating each result sequentially. This suboption is enabled by default on Y-HPC. This suboption is enabled by default on SUSE 9 and Red Hat 4 if you set **-qarch** to **ppc970**. This suboption is enabled on Red Hat 3 U3 if you set **-qarch** to **ppc970** and use **-qenablevmx**. .

Getting the most out of -qhot

Here are some suggestions for using **-qhot**:

- Try using **-qhot** along with **-O2** and **-O3** for all of your code. It is designed to have a neutral effect when no opportunities for transformation exist.
- If you encounter unacceptably long compile times (this can happen with complex loop nests) or if your performance degrades with the use of **-qhot**, try using **-qhot=novector**, or **-qstrict** or **-qcompact** along with **-qhot**.
- If necessary, deactivate **-qhot** selectively, allowing it to improve some of your code.

Related references

- **-qhot** in *XL C/C++ Compiler Reference*
- **-qstrict** in *XL C/C++ Compiler Reference*

Using shared-memory parallelism

Some IBM pSeries™ machines are capable of shared-memory parallel processing. You can compile with **-qsmp** to generate the threaded code needed to exploit this capability. The option implies an optimization level of at least **-O2**.

The following table lists the most commonly used suboptions. Descriptions and syntax of all the suboptions are provided in *XL C/C++ Compiler Reference*.

Table 11. Commonly used **-qsmp** *suboptions*

suboption	Behavior
auto	Instructs the compiler to automatically generate parallel code where possible without user assistance. This is the default setting if you do not specify any **-qsmp** suboptions, and it also implies the **opt** suboption.
omp	Instructs the compiler to observe OpenMP language extensions for specifying explicit parallelism. Note that **-qsmp=omp** is currently incompatible with **-qsmp=auto**.
opt	Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to **-O2 -qhot** in the absence of other optimization options.
fine_tuning	Other values for the suboption provide control over thread scheduling, nested parallelism, locking, etc.

Getting the most out of -qsmp

Here are some suggestions for using the **-qsmp** option:

- Before using **-qsmp** with automatic parallelization, test your programs using optimization and **-qhot** in a single-threaded manner.

- If you are compiling an OpenMP program and do not want automatic parallelization, use **-qsmp=omp:noauto** .
- Always use the reentrant compiler invocations (the *_r* invocations) when using **-qsmp**.
- By default, the run-time environment uses all available processors. Do not set the **XLSMPOPTS=PARTHDS** or **OMP_NUM_THREADS** environment variables unless you want to use fewer than the number of available processors. You might want to set the number of executing threads to a small number or to 1 to ease debugging.
- If you are using a dedicated machine or node, consider setting the **SPINS** and **YIELDS** environment variables (suboptions of the **XLSMPOPTS** environment variables) to 0. Doing so prevents the operating system from intervening in the scheduling of threads across synchronization boundaries such as barriers.
- When debugging an OpenMP program, try using **-qsmp=noopt** (without **-O**) to make the debugging information produced by the compiler more precise.

Related references

- **-qsmp** in *XL C/C++ Compiler Reference*
- "Runtime Options for Parallel Processing" in *XL C/C++ Compiler Reference*
- "OpenMP Runtime Options for Parallel Processing" in *XL C/C++ Compiler Reference*

Using interprocedural analysis

Interprocedural analysis (IPA) enables the compiler to optimize across different files (whole-program analysis), and can result in significant performance improvements. You can specify interprocedural analysis on the compile step only or on both compile and link steps ("whole program" mode). Whole program mode expands the scope of optimization to an entire program unit, which can be an executable or shared object. As IPA can significantly increase compilation time, you should limit using IPA to the final performance tuning stage of development.

You enable IPA by specifying the **-qipa** option. The most commonly used suboptions and their effects are described in the following table. The full set of suboptions and syntax is described in **-qipa** in the *XL C/C++ Compiler Reference*.

Table 12. Commonly used **-qipa** *suboptions*

suboption	Behavior
level=0	Program partitioning and simple interprocedural optimization, which consists of: • Automatic recognition of standard libraries. • Localization of statically bound variables and procedures. • Partitioning and layout of procedures according to their calling relationships. (Procedures that call each other frequently are located closer together in memory.) • Expansion of scope for some optimizations, notably register allocation.
level=1	Inlining and global data mapping. Specifically: • Procedure inlining. • Partitioning and layout of static data according to reference affinity. (Data that is frequently referenced together will be located closer together in memory.) This is the default level if you do not specify any suboptions with the **-qipa** option.

Table 12. Commonly used **-qipa** *suboptions (continued)*

suboption	Behavior
level=2	Global alias analysis, specialization, interprocedural data flow: • Whole-program alias analysis. This level includes the disambiguation of pointer dereferences and indirect function calls, and the refinement of information about the side effects of a function call. • Intensive intraprocedural optimizations. This can take the form of value numbering, code propagation and simplification, code motion into conditions or out of loops, elimination of redundancy. • Interprocedural constant propagation, dead code elimination, pointer analysis, and code motion across functions. • Procedure specialization (cloning).
inline=*variable*	Allows you precise control over function inlining.
fine_tuning	Other values for **-qipa** provide the ability to specify the behavior of library code, tune program partitioning, read commands from a file, etc.

Getting the most from -qipa

It is not necessary to compile everything with **-qipa**, but try to apply it to as much of your program as possible. Here are some suggestions:

- Specify the **-qipa** option on both the compile and link steps of the entire application, or as much of it as possible. Although you can also use **-qipa** with libraries, shared objects, and executables, be sure to use **-qipa** to compile the main and exported functions.
- When compiling and linking separately, use **-qipa=noobject** on the compile step for faster compilation.
- When specifying optimization options in a makefile, remember to use the compiler driver (**xlc**) to link, and to include all compiler options on the link step.
- As IPA can generate significantly larger object files than traditional compilations, ensure that there is enough space in the **/tmp** directory (at least 200 MB), or use the **TMPDIR** environment variable to specify a different directory with sufficient free space.
- Try varying the **level** suboption if link time is too long. Compiling with **–qipa=level=0** can still be very beneficial for little additional link time.
- Use **-qlist** or **-qipa=list** to generate a report of functions that were inlined. If too few or too many functions are inlined, consider using **–qipa=inline** or **–qipa=noinline**. To control inlining of specific functions, use **-Q+** or **-Q-**.

Related references

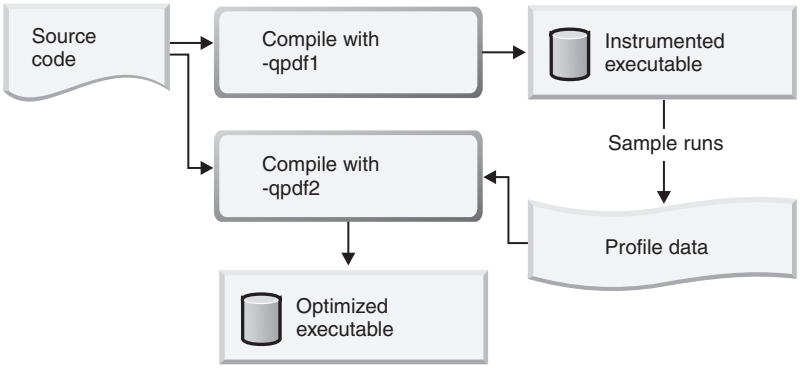
- **-qipa** in *XL C/C++ Compiler Reference*
- **-Q** in *XL C/C++ Compiler Reference*
- **-qlist** in *XL C/C++ Compiler Reference*

Using profile-directed feedback

You can use profile-directed feedback (PDF) to tune the performance of your application for a typical usage scenario. The compiler optimizes the application based on an analysis of how often branches are taken and blocks of code are executed. Because the process 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.

The following diagram illustrates the PDF process.

Figure 2. Profile-directed feedback



You first compile the program with the **-qpdf1** option (with a minimum optimization level of **-O**), which generates profile data by using the compiled program in the same ways that users will typically use it. You then compile the program again, with the **-qpdf2** option. This optimizes the program based on the profile data, by invoking **qipa=level=0**.

Note that you do not need to compile all of the application's code with the **-qpdf1** option to benefit from the PDF process; in a large application, you might want to concentrate on those areas of the code that can benefit most from optimization.

To use the **-qpdf** options:

1. Compile some or all of the source files in the application with **-qpdf1** and a minimum optimization level of **-O**.
2. Run the application using a typical data set or several typical data sets. It is important to use data that is representative of the data that will be used by your application in a real-world scenario. When the application exits, it writes profiling information to the PDF file in the current working directory or the directory specified by the **PDFDIR** environment variable.
3. Compile the application with **-qpdf2**.

You can take more control of the PDF file generation, as follows:

1. Compile some or all of the source files in the application with **-qpdf1** and a minimum optimization level of **-O**.
2. Run the application using a typical data set or several typical data sets. This produces a PDF file in the current directory.
3. Change the directory specified by the **PDFDIR** environment variable to produce a PDF file in a different directory.
4. Re-compile the application with **-qpdf1**.
5. Repeat steps 3 and 4 as often as you want.
6. Use the **mergepdf** utility to combine the PDF files into one PDF file. For example, if you produce three PDF files that represent usage patterns that will occur 53%, 32%, and 15% of the time respectively, you can use this command:

   ```
   mergepdf -r 53 path1  -r 32 path2  -r 15 path3
   ```

7. Compile the application with **-qpdf2**.

To collect more detailed information on function call and block statistics, do the following:

1. Compile the application with **-qpdf1 -qshowpdf -O**.
2. Run the application using a typical data set or several typical data sets. The application writes more detailed profiling information in the PDF file.
3. Use the **showpdf** utility to view the information in the PDF file.

To erase the information in the PDF directory, use the **cleanpdf** utility or the **resetpdf** utility.

Example of compilation with pdf and showpdf

The following example shows how you can use PDF with the **showpdf** utility to view the call and block statistics for a "Hello World" application.

The source for the program file `hello.c` is as follows:

```
#include <stdio.h>
void HelloWorld()
{
printf("Hello World");
}
main()
{
HelloWorld();
return 0;
}
```

1. Compile the source file:

   ```
   xlc -qpdf1 -qshowpdf -O hello.c
   ```

2. Run the resulting program executable **a.out**.
3. Run the **showpdf** utility to display the call and block counts for the executable:

   ```
   showpdf
   ```

The results will look similar to the following:

```
HelloWorld(4):  1 (hello.c)

Call Counters:
5 | 1  printf(6)

Call coverage = 100% ( 1/1 )

Block Counters:
3-5 | 1
6 |
6 | 1

Block coverage = 100% ( 2/2 )

-----------------------------------
main(5):  1 (hello.c)

Call Counters:
10 | 1  HelloWorld(4)

Call coverage = 100% ( 1/1 )

Block Counters:
8-11 | 1
11 |
```

```
Block coverage = 100% ( 1/1 )

Total Call coverage = 100% ( 2/2 )
Total Block coverage = 100% ( 3/3 )
```

Related references

- **-qpdf** in *XL C/C++ Compiler Reference*
- **-showpdf** in *XL C/C++ Compiler Reference*

Other optimization options

The following options are available to control particular aspects of optimization. They are often enabled as a group or given default values when you enable a more general optimization option or level. For more information on these options, see the heading for each option in the *XL C/C++ Compiler Reference*.

Table 13. Selected compiler options for optimizing performance

Option	Description
-qcompact	Suppresses optimizations that would result in larger code size, such as loop unrolling, and function inlining.
-qignerrno	Allows the compiler to assume that **errno** is not modified by library function calls, so that such calls can be optimized. Also allows optimization of square root operations, by generating inline code rather than calling a library function.
-qsmallstack	Instructs the compiler to compact stack storage. Doing so may increase heap usage.
-qinline	Controls inlining of named functions. Can be used at compile time, link time, or both. When **-qipa** is used, **-qinline** is synonymous with **-qipa=inline**.
-qunroll	Independently controls loop unrolling. Is implicitly activated under **-O3**.
-qinlglue	Instructs the compiler to inline the "glue code" generated by the linker and used to make a call to an external function or a call made through a function pointer. 64-bit mode only.
-qtbtable	Controls the generation of traceback table information. 64-bit mode only.
C++ **-qnoeh**	Informs the compiler that no C++ exceptions will be thrown and that cleanup code can be omitted. If your program does not throw any C++ exceptions, use this option to compact your program by removing exception-handling code.
-qnounwind	Informs the compiler that the stack will not be unwound while any routine in this compilation is active. This option can improve optimization of non-volatile register saves and restores. In C++, the **-qnounwind** option implies the **-qnoeh** option.

Summary of options for optimization and performance

The following table presents a summary of the compiler options that deal with optimization and performance tuning. The options are grouped by type. For a description, full syntax, and usage of each option, see the appropriate option heading in the *XL C/C++ Compiler Reference*.

Table 14. Options related to optimization and performance tuning

Optimization flags	Optimization restriction options
-O/-qoptimize **-qagrrcopy**	**-qkeepparm** **-qnoprefetch** **-qstrict** **-qcompact** **-qmaxmem**
Function inlining	**Code size reduction**
-Q **-qinline** **-qinlglue**	**-s** **-qnoeh**
Side effects	**Loop optimization**
-qignerrno **-qisolated_call**	**-qhot** **-qreport** **-qnostrict_induction** **-qunroll**
Whole-program analysis	**Processor and architectural optimization**
-qipa	**-qarch** **-qcache** **-qtune** **-qdirectstorage** **-qenablevmx**
Performance data collection	**Other optimization options**
-p **-qpdf1** **-qpdf2** **-pg** **-qshowpdf**	**-qtocdata** **-qsmallstack** **-qspill**

Chapter 7. Coding your application to improve performance

Chapter 6, "Optimizing your applications," on page 27 discusses the various compiler options that XL C/C++ provides for optimizing your code with minimal coding effort. If you want to take your application a step further, to complement and take the most advantage of compiler optimizations, the following sections discuss C and C++ programming techniques that can improve performance of your code:

Find faster input/output 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 guidelines:

- 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.
- Use **#pragma disjoint** within functions for pointers or reference parameters that can never point to the same memory.
- Declare a nonmember function as static whenever possible. This can speed up calls to the function.
- C++ 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.
- C++ When declaring functions, use the **const** specifier whenever possible.
- C 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 can be passed in appropriate registers.
- C Avoid using unprototyped variable argument functions.
- Design functions so that the most frequently used parameters are in the leftmost 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 non-aggregate 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 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 built-in functions if you include the XL C/C++ header files.

  C Your functions are mapped to built-in 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, or functions called with one or more compile-time constant parameters, especially those that affect **if**, **switch** or **for** statements. You might also want to put these functions into header files, which allows automatic inlining across file boundaries even at low optimization levels. Be sure to inline all functions that only load or store a value, or use simple operators such as comparison or arithmetic operators. Large functions and functions that are called rarely might not be good candidates for inlining.

- Avoid breaking your program into too many small functions. If you must use small functions, seriously consider using the **-qipa** compiler option, which can automatically inline such functions, and uses other techniques for optimizing calls between functions.
- C++ Avoid virtual functions and virtual inheritance unless required for class extensibility. These language features are costly in object space and function invocation performance.

Related references

- **#pragma isolated_call** in *XL C/C++ Compiler Reference*
- **#pragma disjoint** in *XL C/C++ Compiler Reference*
- **-qipa** in *XL C/C++ Compiler Reference*

Manage memory efficiently

Because C++ objects are often allocated from the heap and have limited scope, memory use affects performance more in C++ programs than it does in C programs. For that reason, consider the following guidelines when you develop C++ applications:

- 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.
- Storage pools are a good way of keeping track of used memory (and reclaiming it) without having to resort to an object manager or reference counting.
- 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.
- C++ Use virtual methods only when absolutely necessary.

Optimize variables

Consider the following guidelines:

- 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 can be moved out of loops, and multiple calls with the same parameters can 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.

Related references

- **#pragma isolated_call** in *XL C/C++ Compiler Reference*

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**. This is because **memcpy** copies directly from the source to the destination, while **memmove** might copy the source to a temporary location in memory before copying to the destination (depending on the length of the string).
- 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, whenever possible. This improves certain optimization techniques and reduces memory usage if there are multiple uses of the same string. You can explicitly set strings to read-only by using **#pragma strings (readonly)** in your source files or **-qro** (this is enabled by default) to avoid changing your source files.

Related references

- **#pragma strings (readonly)** in *XL C/C++ Compiler Reference*
- **-qro** in *XL C/C++ Compiler Reference*

Optimize expressions and program logic

Consider the following guidelines:

- 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.
- C++ Use **try** blocks for exception handling only when necessary because they can inhibit optimization.
- Keep array index expressions as simple as possible.

Optimize operations in 64-bit mode

The ability to handle larger amounts of data directly in physical memory rather than relying on disk I/O is perhaps the most significant performance benefit of 64-bit machines. However, some applications compiled in 32-bit mode perform better than when they are recompiled in 64-bit mode. Some reasons for this include:

- 64-bit programs are larger. The increase in program size places greater demands on physical memory.
- 64-bit long division is more time-consuming than 32-bit integer division.
- 64-bit programs that use 32-bit signed integers as array indexes might require additional instructions to perform sign extension each time the array is referenced.

Some ways to compensate for the performance liabilities of 64-bit programs include:

- Avoid performing mixed 32- and 64-bit operations. For example, adding a 32-bit data type to a 64-bit data type requires that the 32-bit type be sign-extended to clear the upper 32 bits of the register. This slows the computation.
- Avoid long division whenever possible. Multiplication is usually faster than division. If you need to perform many divisions with the same divisor, assign the reciprocal of the divisor to a temporary variable, and change all divisions to multiplications with the temporary variable. For example, the function

  ```
  double preTax(double total)
  {
  return total * (1.0 / 1.0825);
  }
  ```

  will perform faster than the more straightforward:

  ```
  double preTax(double total)
  {
  return total / 1.0825;
  }
  ```

  The reason is that the division `(1.0 / 1.0825)` is evaluated once, and folded, at compile time only.
- Use **long** types instead of **signed**, **unsigned**, and plain **int** types for variables which will be frequently accessed, such as loop counters and array indexes. Doing so frees the compiler from having to truncate or sign-extend array references, parameters during function calls, and function results during returns.

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
- IBM (logo)
- PowerPC
- pSeries

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

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).
- The C++ language is also consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:2003 (E)).
- The C and C++ languages are consistent with the OpenMP C and C++ Application Programming Interface Version 2.0.

Program Number: 5724-K77

SC09-7943-01