C++ 4.1

User's Guide

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No: 802-3043-10
Revision A, November 1995

Contents

Tables

Preface

This manual, *C++ 4.1 User's Guide*, is for programmers who are using the C++ programming language. It gives information about C++ and complements the complete C++ documentation set that is described in the"Documentation" section.

Prerequisite Reading

Although there is no required prerequisite reading for this guide, you should have access to good C++ reference books such as *The C++ Programming Language* by Bjarne Stroustrup. You should also have access to the documents described in the following section.

Documentation

C++

Manuals

C++ 4.1 Library Reference Manual

This guide gives information about how to use the following C++ libraries:

- Complex

- Coroutine
- `Iostream`

Installing SunSoft Developer Products on Solaris

This manual tells you how to install C++ software in addition to other software on Solaris.

Tools.h++ Introduction and Reference Manual

The *Tools.h++* Class Library is a set of C++ classes that can greatly simplify your programming while maintaining the efficiency for which C is famous. This manual introduces you to and tells you how to use the Tools.h++ class library.

Articles

"Close as Possible to C, But No Closer"

An article by Andrew Koenig and Bjarne Stroustrup.

"Object Oriented Programming"

An article by Bjarne Stroustrup.

"What Every Computer Scientist Should Know About Floating-Point Arithmetic"

A floating-point white paper by David Goldberg. This paper can be found in the README directory.

On-Line Documentation

AnswerBook Product

Most of the C++ documentation is available through the AnswerBook product. The AnswerBook product provides on-line documentation that enables you to electronically jump from one subject to another and to search for topics by using a word or phrase.

AnswerBook on-line documentation for this product is installed separately. See the *Installing SunPro Software on Solaris* manual for further information on installation.

Error Messages

Error messages give useful information to help you code and debug your program.

Manual Pages (man pages)

Each man page concisely explains a single subject, which could be a user command or library function. Man pages are in the locations shown by Table P-1:

Table P-1 Manual Page Locations

Solaris 2.*x*	*opt install dir*/SUNWspro/SC4.1/man
Solaris 1.*x*	/usr/lang/man
HP-UX	*opt install dir*/SUNWspro/SC4.1/man

Note – Before you use the man command, insert the name of the directory in which you have chosen to install the C++ compiler at the beginning of your search path. Doing this enables you to use the man command. This is usually done in the.cshrc file, in a line with setenv MANPATH= at the start; or in the .profile file, in a line with export MANPATH= at the start.

README *file*

The README file highlights important information about the compiler, including the platforms it supports, the operating environment, documentation, and late-breaking news. View the README file by typing CC -readme.

C++ Migration Guide

The *C++ Migration Guide* helps you migrate your code from cfront-based C++ 3.0 to the current compiler. This manual is displayed when you type CC -migration.

Solaris Documentation

These manuals are available to you on-line via the AnswerBook™ product:

Programming Utilities and Libraries

The *Programming Utilities and Libraries* manual covers a few of the tools that can aid you in programming. These include:

`lex` — Generates programs used in simple lexical analysis of text, solves problems by recognizing different strings of characters.

`yacc` — Takes a description of a grammar and generates a C function to parse the input stream according to that grammar.

`prof` — Produces an execution profile of the modules in a program.

`make` — Automatically maintains, updates, and regenerates related programs and files.

System V `make` — Describes a version of `make` that is compatible with older versions of the tool.

`sccs` — Allows you to control access to shared files and to keep a history of changes made to a project.

`m4` — Macro language processor.

This manual is bundled with the operating system documentation.

Solaris Linker and Libraries Manual

The *Solaris Linker and Libraries Manual* gives information on linking libraries.

Commercially Available Books

The following is a partial list of available books on C++.

A C++ Primer, 2nd Ed, Stanley B. Lippman (Addison-Wesley, 1989)

A Guide to Object-Oriented Programming in C++, Keith Gorlen (John Wiley & Sons)

C++ for C Programmers, Ira Pohl (Benjamin/Cummings, 1989)

The Annotated C++ Reference Manual, Margaret A. Ellis and Bjarne Stroustrup (Addison-Wesley, 1990)

The C++ Programming Language, 2nd Ed, Bjarne Stroustrup (Addison-Wesley, 1991)

Object-Oriented Design with Applications, 2nd Ed, Grady Booch (Addison-Wesley)

Effective C++—50 Ways to Improve Your Programs and Designs, Scott Meyers (Addison-Wesley)

Periodicals

The following is a partial list of the many periodicals that include articles about C++.

- *Dr. Dobbs Journal*
- *The C++ Report*
- *Object Magazine*
- *Journal of Object-Oriented Programming*

What Typographic Changes and Symbols Mean

The following table describes the typographic conventions and symbols used in this book.

Table P-2 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. system% You have mail.
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
♦	A single-step procedure	♦ **Click on the Apply button.**

Code samples are included in boxes and may display the following:

Table P-2 Typographic Conventions *(Continued)*

Typeface or Symbol	Meaning	Example
`%`	C shell prompt	`system%`
`$`	Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`
`[ ]`	Square brackets contain arguments that can be optional or required.	`-d[y|n]`
`|`	The “pipe” or “bar” symbol separates arguments, only *one* of which may be used at one time.	`-d[y|n]`
`,`	The comma separates arguments, *one or more* of which may be used at one time.	`-xinline=[`*f1,...fn*`]`
`:`	The colon, like the comma, is sometimes used to separate arguments.	`-R`*dir*`[:`*dir*`]`
`...`	The ellipsis indicates omission in a series.	`-xinline=[`*f1,...fn*`]`
`%`	The percent sign indicates the word following it has a special meaning.	`-ftrap=%all`
`<>`	In ASCII files, such as the `README` file, angle brackets contain a variable that must be replaced by an appropriate value.	`-xtemp=<dir>`

Introduction 1

This chapter provides a brief conceptual overview of C++, with particular emphasis on the areas of difference and similarity with C. Chapter 5, “Using C and C++,” summarizes issues important to C programmers moving to C++.

The Compiler Package

The compiler package includes:

- The compiler front end, `ccfe`
- The driver `CC`
- On-line `README` files containing the latest known software and documentation bugs and other pertinent information.
- Manual pages, also known as man pages
- The C++ name demangling tool set (`dem` and `c++filt`)
- C++ library functions for stream I/O, complex arithmetic, tasking, and other operations.
- `Tools.h++` class library
- `#include` files you can use for standard features such as signals and ctype with C++ programs
- The template pre-linker, `tdb_link`

1

C++ Tools

Most of the C++ tools are now incorporated into traditional UNIX tools. These tools are bundled with operating system. They are:

- `lex`
- `yacc`
- `prof`
- `gprof`
- `nm`
- `ctags`
- `rpcgen`

Please see the *Profiling Tools* manual and associated man pages for further information on these UNIX tools.

The C++ Language

This version of C++ supports the C++ language as described in *The C++ Programming Language* by Margaret Ellis and Bjarne Stroustrup, with a few deletions and a number of extensions.

C++ is designed as a superset of the C programming language. While retaining the C facility of efficient low-level programming, C++ adds:

- Stronger type checking
- Extensive data abstraction features
- Support for object-oriented programming

This last feature, particularly, allows good design of modular and extensible interfaces among program modules.

Compatibility with C

C++ is almost entirely compatible with C. The language was purposely designed this way; C programmers can learn C++ at their own pace and incorporate features of the new language when it seems appropriate. What is new about C++ supplements what is good and useful about C; most importantly, C++ retains C's efficient interface to the hardware of the computer, including types and operators that correspond directly to components of computing equipment.

C++ does have some important differences with C; an ordinary C program may not be accepted by the C++ compiler without some modifications. Chapter 5, “Using C and C++,” discusses what you must know to move from programming in C to programming in C++.

Even though the differences between C and C++ are most evident in the way you can design interfaces between program modules, C++ retains all of C’s facilities for designing such interfaces. You can, for example, link C++ modules to C modules, so you can use C libraries with C++ programs.

Type Checking

A compiler or interpreter performs *type checking* when it ensures that operations are applied to data of the correct type. C++ has stronger type checking than C, though not as strong as that provided by Pascal. The approach to type checking is different from the approach in languages like Pascal: where Pascal always prohibits attempts to use data of the wrong type, the C++ compiler produces errors in some cases, but in others converts data to the correct type.

Rather than having the compiler do these automatic conversions, you can explicitly convert between types, as you can in C.

A related area involves overloaded function names. In C++, you can give any number of functions the same name. The compiler decides which function should be called by checking the types of the parameters to the function call. This action may lead to ambiguous situations. If the resolution is not clear at compile-time, the compiler issues an “ambiguity” error.

Integer Constant Type

The type of an integer constant is determined by its value and suffix.The following rules are analogous to those in ANSI-C, extended to include `long long` and unsigned `long long`. The constant has the type of the first type in the list that fits, in each of the following cases.

1. Decimal and no suffix: `int, long, unsigned long, long long, unsigned long long.`

2. Octal or hexadecimal and no suffix: `int, unsigned int, long, unsigned long, long long, unsigned long long`.

3. Suffixed by u or U: `unsigned int, unsigned long, unsigned long long.`

4. Suffixed by l or L: `long, unsigned long, long long, unsigned long long.`

5. Suffixed by (u or U) and (l or L): `unsigned long, unsigned long long.`

6. Suffixed by ll or LL: `long long, unsigned long long.`

7. Suffixed by ull or ULL: `unsigned long long.`

For example, 2147483648, (INT_MAX+1) and -2147483648 are both unsigned `long`, instead of `long long`. This is to preserve the semantics of the application where long long is not available. This can sometimes lead to unexpected behavior, for instance in "double d = -2147483648;", d results being positive.

Classes and Data Abstraction

A class is a user-defined type. If you are a C programmer, a class is an extension of the idea of `struct` in C. Like the predefined types (but unlike user-defined types in languages like Pascal), classes are defined not only with data storage but also with operations that apply to the data. In C++, these operations include operators and functions. For example, if you define a class `carrot`, you can define the + operator so it has a meaning when used with `carrots`. If `carrot1` and `carrot2` are objects of the type `carrot`, then the expression:

```
carrot1 + carrot2
```

has a value determined by your definition of + in the case of `carrots`. This definition does not override the original definition of +; as with overloaded function names, the compiler determines from context what definition of + it should use. Operators with extra definitions like this are called *overloaded operators*.

In addition to operators, classes may have member functions, functions that exist to operate on objects of that class.

C++ provides classes as a means for *data abstraction*. Programs that are designed with data abstraction are designed by deciding what types (classes) you want for the program's data and then deciding what operations each type needs.

The members of a class can be divided into three parts, `public`, `private`, and `protected`. The `public` part is available to any function; the `private` part is available only to member and friend functions; the `protected` part is available to members, friends, and members of derived classes.

Object-Oriented Features

A program is object-oriented when it is designed with classes, and the classes are organized so that common features are embodied in base classes, sometimes called parent classes. The feature that makes this possible is inheritance. A class in C++ can inherit features from one base class or from several. A class that has a base class is said to be derived from the base class.

Other Differences from C

C++ differs from C in a number of other details. In C++:

- You can use defined constants to avoid using the preprocessor to use named constants in your program.
- You must use function prototypes; they are optional in C.
- Free store operators `new` and `delete` create dynamic variables.
- References, alternate "handles" on the same object, are automatically dereferenced pointers, and act like an alternate name for a variable. You can use references as function parameters.
- Functional syntax for type coercions is supported.
- Programmer-defined automatic type conversion is allowed.
- Variable declarations are allowed anywhere, not just at the beginning of the block.
- A new comment delimiter begins a comment that continues to the end of the line.
- The name of an enumeration or class is also automatically a type name.

- You can assign default values to function parameters.
- You can use inline functions to ask the compiler to replace a function call with the function body, improving program efficiency.

Native Language Support

This release of C++ supports the development of applications in languages other than English, including most European languages and Japanese. As a result, you can easily switch your application from one native language to another. This feature is known as internationalization.

In general, the C++ compiler implements internationalization as follows:

- C++ recognizes ASCII characters from international keyboards (in other words it has keyboard independence and is 8-bit clean).
- C++ allows the printing of some messages in the native language.
- C++ allows native language characters in comments, strings, and data.

Compilers present unique problems for internationalization because they have qualities that other software products do not. An internationalized C++ compiler does not allow input and output in the various international formats. If it did, it would not comply with the language standard appropriate for its language.

For example, some standards specify a period (`.`) as the decimal unit in the floating point representation. Consider the following program:

```
#include <stream.h>
main()
{
    float r = 1.2;
    cout << r << "\n";
}
```

Here is the output when you compile the program on the internationalized C++ compiler:

```
1.2
```

If you reset your system locale to, say, France and rerun the program, you still receive the same output. The period is not replaced with a comma, the French decimal unit.

Variable names cannot be internationalized, and must be in the English character set.

You can change your application from one native language to another by setting the locale. For information on this and other native language support features, see the operating system documentation.

1

The Compiler 2

This chapter demonstrates how to compile and link your code, as well as how to use compiler pragmas and options.

Compiling

Before using the `CC` command, insert the name of the directory in which you have chosen to install the C++ compiler at the beginning of your search path. The default paths are:

Solaris 1.*x*	`/usr/lang/SC4.0`
Solaris 2.*x and HP-UX*	`/opt/SUNWspro/SC4.0`

To compile a simple program, `myprog`, enter the following command:

```
% CC myprog.cc -o myprog
```

The resulting executable file is called `myprog` because this command-line uses the `-o` *name* argument. Without that argument, the executable file has the default name, `a.out`.

The possible file name extensions for the source file are: `.c`, `.C`, `.cc`, `.cpp`, or `.cxx`.

2

Compiling and Linking

The sample program `testr`, as used in Appendix , “Code Samples”, consists of two modules: the main program module, `testr.cc`, and the string class module, `str.cc` and `str.h`.

When you have a second module like the string class module, both the implementation part of the second module and the main program module must include the header file for the second module. For example, `testr.cc` and `str.cc` include the header file, `str.h`, with a line like:

```
#include "str.h"
```

If there is no object file for the second module, you can compile the second module and link it with the program with a command-line like:

```
% CC testr.cc str.cc -o testr
```

Alternately, you can create an object file for the second module with this command-line:

```
% CC -c str.cc
```

This line does not invoke the linker and results in an object file called `str.o`.

When there is an object file for the unit, you can compile the program and link it with the unit, as follows:

```
% CC str.o test.cc -o testr
```

Compatibility between C++ 4.0.1 and C++ 4.1

The object files produced by C++ 4.1 and C++ 4.0.1 are binary compatible for most cases. That is, object files created by C++ 4.0.1 can be linked using the C++ 4.1 compiler, and object files created by C++ 4.1 can be linked using the C++ 4.0.1 compiler without any problems. However, this may not be true if you are using exceptions and link statically with `libC`. Note the following cases:

1. If you are not using exceptions, objects files produced by the two compilers are binary compatible. You can create binary files with either compiler, and link using either compiler, without any compatibility problems. This holds true whether you link statically or dynamically with `libC.`
2. If you are using exceptions, the result will depend on whether you are linking statically or dynamically with `libC`.

a. If you link statically with `libC`, the object files produced by C++ 4.1 are not binary compatible wth the `libC.a` of C++ 4.0.1. That is, if you create binary files with C++ 4.1, and link them using the C++ 4.0.1 compiler, the resulting executable may not work. You may get a core dump. However, binary files created with the C++ 4.0.1 compiler will work with the `libC.a` of C++ 4.1. If you create binary files with the C++ 4.0.1 compiler, and link them using the C++ 4.1 compiler, there will be no compatibility problems.

b. If you link dynamically with `libC`, and are running Solaris 2.4 or earlier version of the operating system, you must install `libC.so.5` patch 101242-10 on your machine. Once you have installed this patch, binaries created by the two compilers will be compatible. You can create binary files with either compiler, and link using either compiler, without any compatibility problems.

c. If you link dynamically with `libC` and are running Solaris 2.5 on your machine, you don't need to install the `libC.so.5` patch. Binaries created by the two compilers will be compatible. (You can run `ldd` on your executable to find out if you have linked dynamically with `libC.so.5`.)

Multiple File Extensions in `make`

The C++ compiler accepts file extensions other than `.cc`. You can incorporate different file extensions (suffixes) into C++ in two ways: by adding them to the system default makefile or to your makefile.

Suffix Additions to the System Default Makefile

You can add suffixes to C++ by adding them to the system default makefile. This file is, depending on your operating environment:

Solaris 1.*x*	`/usr/include/make/default.mk`
Solaris 2.*x*	`/usr/share/lib/make/make.rules`
HP-UX	`/usr/share/lib/make/make.rules`

Here is an example. Become `root` to make these changes:

1. Become root.

2. **If you use Solaris 1.*x*, add `.C .C~` to the end of the `SUFFIXES` macro. If you use Solaris 2.*x*, `.C` and `.C~` are already in the `SUFFIXES` macro.**

3. **Add these lines to the end of the system default makefile.**
   The indented lines are tabbed.

```
.C:
    $(LINK.cc) -o $@ $< $(LDLIBS)
.C.o:
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
.C.a:
    $(COMPILE.cc) -o $% $<
    $(AR) $(ARFLAGS) $@ $%
    $(RM) $%
```

Note – Since `.c` is supported as a C-language suffix, it is the one suffix that cannot be added to the `SUFFIXES` macro to support C++. Write explicit rules in your own makefile to handle C++ files with a `.c` suffix.

Suffix Additions to Your Makefile

The following example adds `.C` as a valid suffix for C++ files.

1. **Add the `SUFFIXES` macro to your makefile:**

   ```
   .SUFFIXES: .C .C~
   ```

   This line can be located anywhere in the makefile.

2. **Add these lines to your makefile:**
   The indented lines are tabbed.

```
.C:
    $(LINK.cc) -o $@ $< $(LDLIBS)
.C.o:
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
.C.a:
    $(COMPILE.cc) -o $% $<
    $(AR) $(ARFLAGS) $@ $%
    $(RM) $%
```

Components

`CC` uses the following components to compile C++ source code to object code.

For SPARC platforms:

1. `ccfe` **performs preprocessing and compilation.**
2. `iropt` **optimizes for execution time. This step is optional; any level of optimization invokes this step.**
3. `cg` **performs code generation when optimization is specified.**
4. `tdb_link` **performs template instantiation of out-of-date templates, and invokes the linker.**
5. `ld` **performs link editing.**
6. `cdlink` ***(Solaris 1.x)*** **performs post-link operations to handle static constructors and destructors.**

For x86, PowerPC and HP-UX platforms:

1. `ccfe` **performs preprocessing and compilation.**
2. `cg386` **(*on X86*)** `cgpa` **(*on HP-UX*)** `cgppc` **(*on PowerPC*) prepares intermediate code for code generation.**
3. `codegen` **performs optimizations and code generation.**
4. `fbe` **(*on X86 and PowerPC*) or** `gas` **(*on HP-UX*) generates** `.o` **file from** `.s` **assembly file.**
5. `tdb_link` **performs template instantiation of out-of-date templates, and invokes the linker.**
6. `ld` **performs link editing.**
7. `cdlink` ***(HP-UX)*** **performs post-link operations to handle static constructors and destructors.**

2

Pragmas

A pragma, also called a compiler directive, is a special comment that directs the compiler to do something. Preprocessing lines in the following example specify implementation-defined actions.

`#pragma` *preprocessor-tokens*

The pragmas discussed in this section are recognized in the compilation system. The compiler ignores unrecognized pragmas.

Note – `#pragma` in template definition files are ignored.

`#pragma align`

(Solaris 2.x only) Use `#pragma align` *integer*`(`*variable*`[,`*variable*`]...)`to make the parameter variables memory-aligned to *integer* bytes, overriding the default. The following limitations apply:

- *integer* must be a power of 2 between 1 and 128; valid values are: 1, 2, 4, 8, 16, 32, 64, and 128.
- *variable* is a global or static variable; it cannot be a local variable.
- If the specified alignment is smaller than the default, the default is used.
- The `pragma` line must appear before the declaration of the variables which it mentions; otherwise, it is ignored.
- Any variable mentioned but not declared in the code following the pragma line is ignored. For example:

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct S {int a; char *b;} astruct;
```

`#pragma init` *and* `#pragma fini`

(Solaris 2.x only) Use `#pragma init(`*identifier* `[ ,` *identifier* `] ...)`to mark *identifier* as an initialization function. Such functions are expected to be of type `void`, to accept no arguments, and to be called while constructing the memory

image of the program at the start of execution. In the case of initializers in a shared object, they are executed during the operation that brings the shared object into memory, either program start-up or some dynamic loading operation, such as `dlopen()`. The only ordering of calls to initialization functions is the order in which they are processed by the link editors, both static and dynamic.

Within a source file, the functions specified in `#pragma init` are executed after the static constructors in that file. You must declare the identifiers before using them in the `#pragma`.

(Solaris 2.x only) Use `#pragma fini (`*identifier* `[, `*identifier* `]...)` to mark *identifier* as a finalization function. Such functions are expected to be of type `void`, to accept no arguments, and to be called either when a program terminates under program control, or when the containing shared object is removed from memory. As with initialization functions, finalization functions are executed in the order processed by the link editor.

In a source file, the functions specified in `#pragma fini` are executed after the static destructors in that file. You must declare the identifiers before using them in the `#pragma`.

`#pragma ident`

(Solaris 2.x only) Use `#pragma ident` *string* to place *string* in the `.comment` section of the executable.

`#pragma unknown_control_flow`

Use `#pragma unknown_control_flow (`*name*`, [,`*name*`] ...)` to specify a list of routines that violate the usual control flow properties of procedure calls. For example, the statement following a call to `setjmp()` can be reached from an arbitrary call to any other routine. The statement is reached by a call to `longjmp()`.

Since such routines render standard flowgraph analysis invalid, routines that call them cannot be safely optimized; hence, they are compiled with the optimizer disabled.

#pragma weak

(Solaris 2.x only) Use #pragma weak to define a weak global symbol. This pragma is used mainly in source files for building libraries. The linker does not warn you if it cannot resolve a weak symbol. The line:

#pragma weak *function_name*

defines *function_name* to be a weak symbol. No error messages are generated if the linker cannot find a definition for *function_name*. The line:

#pragma weak *function_name1* = *function_name2*

defines *function_name1* to be a weak symbol, an alias for the symbol *function_name2*. You must declare *function_name1* and *function_name2* before using them in the pragma. For example:

```
extern void bar(int)
extern void _bar(int)
#pragma weak _bar=bar
```

The rules are:

- If your program calls but does not define *function_name1*, the linker uses the definition from the library.
- If your program defines its own version of *function_name1*, then the program definition is used, and the weak global definition of *function_name1* in the library is not used.
- If the program directly calls *function_name2*, the definition from the library is used; a duplicate definition of *function_name2* causes an error.

Options

This section describes the C++ compiler options, arranged alphabetically. These descriptions are also available in the man page, CC(1).

In addition, these descriptions are summarized in:

- The *C++ 4.1 Quick Reference*, a document provided with this compiler
- The -flags or -help option

Note – In the descriptions are many references to the linker, `ld`. For details about `ld`, see the `ld`(1) man page.

`ccfe` *and* `ld`

You can pass command-line options to the following programs:

- The preprocessor and compiler `ccfe`; see the `CC`(1) man page
- The linker `ld`; see the `ld`(1) man page

The options for `ccfe` and `ld` do not conflict.

`-386`

(x86) Directs the compiler to generate code for the best performance on the Intel 80386 microprocessor.

See also “-native” on page 32 and “-p” on page 38.

`-486`

(x86) Directs the compiler to generate code for best performance on the Intel 80486 microprocessor.

See also “-native” on page 32 and “-p” on page 38.

`-a`

(SPARC) (x86) (PowerPC) Prepares object code for coverage analysis, using `tcov`. This option helps you analyze your program at runtime, but is incompatible with `-g`.

This option is the old style of basic block profiling for `tcov`. See `-xprofile=tcov` for information on the new style of profiling, the `tcov`(1) man page, and the *Profiling Tools* manual for more details.

If set at compile-time, the `TCOVDIR` environment variable specifies the directory where the .d files are located. If this variable is not set, then the .d files remain in the same directory as the .f files.

The -xprofile=tcov and the -a options are compatible in a single executable. That is, you can link a program that contains some files which have been compiled with -xprofile=tcov, and others with -a. You cannot compile a single file with both options.

-Bbinding

Specifies whether library bindings for linking are dynamic (shared) or static (nonshared). The values for *binding* are static and dynamic. -Bdynamic is the default. You can use the -B option to toggle several times on a command-line.

For more information on this option on SPARC and x86 platforms, see the ld(1) man page and the Solaris documentation.

-Bdynamic

Directs the link editor to look for lib*lib*.so files on Solaris, then on HP-UX to look for lib*lib*.sl files. Use this option if you want shared library bindings for linking. If the lib*lib*.so files are not found, it looks for lib*lib*.a files. When you use this option on HP-UX, the compiler passes the -a shared option to ld. The CC driver links the following libraries by default:

Solaris 2.*x*	-lsunmath -lm -lC -lC_mtstubs -lw -lcx -lc
Solaris 1.*x*	-lsunmath -lm -lC -lansi -lcx -lc
HP-UX	-lsunmath -lm -lC -lcx -lc -lPW

By default, the CC driver passes the following -l options to ld:

-lsunmath -lm -lC -lC_mtstubs -lw -lcx -lc

For more details on -l, refer to "-l*lib*" on page 31.

To link some of these libraries statically, use -nolib, as described in "-nolib" on page 33.

-Bstatic

Directs the link editor to look *only* for files named `lib`*lib*`.a`. The `.a` suffix indicates that the file is `static`, that is, nonshared. Use this option if you want nonshared library bindings for linking. When you use this option on HP-UX, the compiler passes the `-a archive` option to `ld`.

On Solaris, this option and its arguments are passed to the linker, `ld`.

-bsdmalloc

(Solaris 1.x) Uses the faster `malloc` from the library `libbsdmalloc.a`. This `malloc` is faster, but uses more memory.

-c

Directs the `CC` driver to suppress linking with `ld` and produces a `.o` file for each source file. If you specify only one source file on the command-line, then you can explicitly name the object file with the `-o` option. For example:

- If you enter **`CC -c x.cc`**, the object file, `x.o`, is generated.
- If you enter **`CC -c x.cc -o y.o`**, the object file, `y.o`, is generated.

See also “-o filename” on page 37.

-cg[89|92]

–cg89

(SPARC) This is a macro for:
`-xarch=v7 -xchip=old -xcache=16/64/4:1024/64/1.`

–cg92

(SPARC) This is a macro for:
`-xarch=v8 -xchip=super -xcache=64/32/1.`

+d

Prevents the compiler from expanding inline functions. This option is turned on when you specify `-g`, the debugging option.

The debugging option, `-g0`, does not turn on `+d`. See "-g0" on page 27.

-D*name*[=*def*]

Defines a macro symbol *name* to the preprocessor. Doing so is equivalent to including a `#define` directive at the beginning of the source. If you do not use =*def*, *name* is defined as `1`. You can use multiple `-D` options.

The following values are predefined:

- `__BUILTIN_VA_ARG_INCR` (for `__builtin_alloca` `__builtin_va_alist` `__builtin_va_arg_incr` keywords in `varargs.h`, `stdarg.h`, and `sys/varargs.h`)
- `__cplusplus`
- `__STDC__`
- `__sun`
- `sun`
- `__SUNPRO_CC=0x410`
- `__`<*uname* `-s`>`_`<*uname* `-r`> (replaces invalid characters with underscores, as in –D__SunOS_5_3; –D__SunOS_5_4); –D__HP_UX_A_09_05
- `__unix`
- `unix`
- `_WCHAR_T_`
- `_SIGNEDCHAR_`

HP-UX only:

- `__hp9000S700`
- `hp9000S800`
- `__hp9000S800`
- `__hppa`
- `hppa`
- `__hpux`
- `hpux`
- `__HPUX_SOURCE`

Solaris 2.x only:

- `__SVR4`

SPARC only:

- `__sparc`
- `sparc`

x86 only:

- `__i386`
- `i386`

PowerPC only:

- `__PPC`

The macros, `sparc`, `unix`, `sun`, and `i386`, are not defined if `+p` is used. The value of `__SUNPRO_CC` indicates the release number of the compiler. You can use these values in such preprocessor conditionals as: `#ifdef __sparc`.

-d[y/n]

(Solaris 2.x)

`-dy` specifies dynamic linking, which is the default, in the link editor.

`-dn` specifies static linking in the link editor.

This option and its arguments are passed to `ld`.

-dalign

(SPARC) Generates `double-word load` and `store` instructions whenever possible for improved performance. This option assumes that all `double` type data are `double`-word aligned.

-dryrun

Displays what options the driver has passed to the compiler. This option directs the driver `CC` to show, but not execute, the commands constructed by the compilation driver.

-E

Directs the CC driver to run only the preprocessor on C++ source files, and to send the result to stdout (standard output).

+e[0|1]

Controls virtual table generation, as follows:

- +e0 suppresses the generation of virtual tables, and creates external references to those that are needed.
- +e1 creates virtual tables for all defined classes with virtual functions. When you compile with this option, also use the -noex option; otherwise, the compiler generates virtual tables for internal types used in exception handling.

-fast

Selects a combination of compilation options for optimum execution speed. This option provides near maximum performance for most applications by choosing the following compilation options:

- Fastest code-generation option available on the compile-time hardware
- Optimization level –02
- Set of inline expansion templates
- The –fns option *(SPARC only)*
- The –ftrap=*common* option *(SPARC only)*
- The –fnonstd floating-point option *(non-SPARC platforms only)*
- The –dalign option *(SPARC only)*
- The -fsimple=1 option *(SPARC only)*

If you combine –fast with other compilation options, the -dalign option applies; see “-dalign” on page 21.

The code generation option, the optimization level, and use of inline template files can be overridden by subsequent switches. For example, although the optimization part of –fast is –02, the optimization part of -fast -03 is -03. -fast brings in -fsimple; see “-fsimple” on page 24.

Do not use this option for programs that depend on IEEE standard exception handling; different numerical results, premature program termination, or unexpected `SIGFPE` signals may occur.

Note – The criteria for the `-fast` option vary with the compilers from SunSoft: C, C++, FORTRAN 77, and Pascal. Please see the appropriate documentation for the specifics.

The `-fast` option includes `-fns -ftrap=%none`; that is, turn off all trapping. In previous SPARC releases, the `-fast` macro option included `-fnonstd;` now it does not. In x86, PowerPC, and HP releases, the `-fast` macro option still includes `-nonstd.`

The `-fast` macro includes `-native` in its expansion.

-flags

Displays a brief description of each compiler option. Also displayed are: phone numbers to call to obtain additional information on SunSoft products, for technical support, and information on how to send comments on Sun products.

-fnofma

(PowerPC) Causes the code generator to avoid producing fused multiply-add and fused multiply-subtract instructions. These instructions can produce slightly different results than a multiply-add or multiply-subtract sequence. This flag is only needed for programs which are very sensitive to floating point rounding.

-fnonstd

Causes nonstandard initialization of floating-point arithmetic hardware. In addition, the `–fnonstd` option causes hardware traps to be enabled for floating-point overflow, division by zero, and invalid operations exceptions. These are converted into SIGFPE signals; if the program has no SIGFPE handler, it terminates with a memory dump. See the `ieee_handler`(3m) man page for more information.

By default, IEEE 754 floating-point arithmetic is nonstop, and underflows are gradual.

This option is a synonym for `–fns –ftrap=common`.

`-fns`

(*Solaris 2.x*) (*SPARC*) Turns on the SPARC non-standard floating-point mode. The default is the SPARC standard floating-point mode.

If you compile one routine with `-fns`, then compile all routines of the program with the `–fns` option; otherwise, you can get unexpected results.

`-fround=`*r*

(*Solaris 2.x*) (*SPARC*) Sets the IEEE 754 rounding mode.

r must be one of: `nearest`, `tozero`, `negative`, `positive`.

The default is `-fround=nearest`.

This option sets the IEEE 754 rounding mode that:

- Can be used by the compiler in evaluating constant expressions.
- Is established at runtime during the program initialization.

The meanings are the same as those for the `ieee_flags` subroutine.

If you compile one routine with `-fround=`*r*, compile all routines of the program with the same `–fround=`*r* option; otherwise, you can get unexpected results.

`-fsimple`

(Not supported on HP-UX)

Optimizes mathematically equivalent expressions. The optimizer acts as if a simple floating-point model holds during compilation and runtime, and it is allowed to optimize without regard to roundoff or numerical exceptions. The optimizer can assume the following:

- The IEEE 754 default rounding and trapping models hold.
- No exceptions arise other than inexact.

- The program does not test for inexact exceptions.
- There are no infinities or NaNs.
- The program does not depend on distinguishing by the sign of zero.

-fstore

(x86) Causes the compiler to convert the value of a floating point expression or function to the type on the left hand side of an assignment, when that expression or function is assigned to a variable, or when the expression is cast to a shorter floating point type rather than leaving the value in a register. Due to roundoffs and truncation, the results may be different from those that are generated from the register values. This is the default mode.

To turn off this option, use the `-nofstore` option.

-ftrap=*t*

(*Solaris 2.x) (SPARC*) Sets the IEEE 754 trapping mode.

t is a comma-separated list that consists of one or more of the following: `%all`, `%none`, `common`, [no%]`invalid`, [no%]`overflow`, [no%]`underflow`, [no%]`division`, [no%]`inexact`.

The default is `-ftrap=%none`.

This option sets the IEEE 754 trapping modes that are established at program initialization. Processing is left-to-right. The common exceptions, by definition, are invalid, division by zero, and overflow.

Example: `-ftrap=%all,no%inexact` means set all traps, except `inexact`.

The meanings are the same as for the `ieee_flags` subroutine, except that:

- `%all` turns on all the trapping modes.
- `%none`, the default, turns off all trapping modes.
- A `no%` prefix turns off that specific trapping mode.

If you compile one routine with `-ftrap=`*t*, compile all routines of the program with the same `-ftrap=`*t* option; otherwise, you can get unexpected results.

-G

Instructs the linker to build a shared library; see the ld(1) man page and the *C++ 4.1 Library Reference Manual*. All source files specified in the command-line are compiled with -pic.

Use this option when building shared libraries of templates. Any generated templates are then automatically included in the library.

On Solaris 2.x, the following text options are passed to ld if -c is not specified:

- -dy
- -G
- -ztext

On Solaris 1.x and HP-UX, use this option when building shared libraries. E

Note – Exceptions are not supported in shared libraries on HP-UX.

-g

Instructs both the compiler and the linker to prepare the file or program for debugging. The tasks include:

- Producing more detailed information, known as stabs, in the symbol table of the object files and the executable
- Producing some “helper functions,” which the Debugger can call to implement some of its features
- Disabling the inline generation of functions; that is, using this option implies the +d option as well
- Disabling certain levels of optimization. You can use this option along with -O for the optimization level that you desire.

On SPARC, this option makes -xildon the default incremental linker option. See“-xildon and -xildoff. Invokes ild in place of ld unless any of the following are true:

- The -G option is present
- The -xildoff option is present
- Any source files are named on the command line

See also the descriptions for “-g0” and “+d,” as well as the ld(1) man page. The dbx *User’s Guide* provides details about stabs and “lazy stabs.”

-g0

Instructs the compiler to prepare the file or program for debugging, but *not* to disable inlining.

For details about the +d option, see “+d” on page 20.

-H

Prints, one per line, the path name of each #include file included during the current compilation on the standard error output.

-help

Same as -flags, as described in “-flags” on page 23.

-h*name*

(Solaris 2.x) (x86) (PowerPC) Names a shared dynamic library and provides a way to have versions of a shared dynamic library.

This is a loader option, passed to ld. In general, the name after -h should be exactly the same as the one after -o. A space between the -h and *name* is optional.

The compile-time loader assigns the specified name to the shared dynamic library you are creating. It records the name in the library file as the intrinsic name of the library. If there is no -h*name* option, then no intrinsic name is recorded in the library file.

Every executable file has a list of needed shared library files. When the runtime linker links the library into an executable file, the linker copies the intrinsic name from the library into that list of needed shared library files. If there is no intrinsic name of a shared library, then the linker copies the path of the shared library file instead. This command-line is an example:

```
% CC -G -o libx.so.1 -h libx.so.1 a.o b.o c.o
```

–i

(Solaris 2.x) Is passed to linker to ignore any LD_LIBRARY_PATH setting.

-inline=*rlst*

(Not supported on HP-UX) Inlines the routines that you specify in the *rlst* list—a comma-separated list of functions and subroutines. This option is used by the optimizer.

Note – This option does not affect C++ inline functions, and is not related to the +d option.

If a function specified in the list is not declared as `extern "C"`, the function name should be mangled. You can use the `nm` command on the executable file to find the mangled function names. For functions declared as `extern "C"`, the names are not mangled by the compiler.

If you compile with the flag `-O3`, using this option can increase optimization by restricting inlining to only those routines in the *rlst* list.

If you compile with the flag `-O4`, the compiler tries to inline all user-written subroutines and functions.

A routine is not inlined if any of the following conditions apply. No warning is issued.

- Optimization is less than `-O3`.
- A routine cannot be found.
- The compiler does not consider inlining the routine an advantage or safe.
- The source for the routine is not in the file being compiled.

-I*pathname*

Adds *pathname* to the list of directories that are searched for `#include` files with relative file names—those that do not begin with a slash. The preprocessor searches for `#include` files in this order:

1. For files included in `"..."`, in the directory containing the source file

For files included in `<...>`, the directory containing the source file is not searched.

2. In the directories named with `-I` options, if any
3. In the standard directory for C++ header files:

Solaris 2.*x and HP-UX*	`/opt/SUNWpro/SC4.0/include/CC`
Solaris 1.*x* *(SunOS 4.1.1)*	`/usr/lang/SC4.0/include/CC_411`
(SunOS 4.1.2)	`/usr/lang/SC4.0/include/CC_412`
(SunOS 4.1.3)	`/usr/lang/SC4.0/include/CC_413`
(SunOS 4.1.3_U1)	`/usr/lang/SC4.0/include/CC_413_U1`

4. In the standard directory for ANSI C header files:

Solaris 2.*x and HP-UX*	`/opt/SUNWpro/SC4.0/include/cc`
Solaris 1.*x*	
(SunOS 4.1.1)	`/usr/lang/SC4.0/include/cc_411`
(SunOS 4.1.2)	`/usr/lang/SC4.0/include/cc_412`
(SunOS 4.1.3)	`/usr/lang/SC4.0/include/cc_413`
(SunOS 4.1.3_U1)	`/usr/lang/SC4.0/include/cc_413_U1`

5. In `/usr/include`

-keeptmp

Retains the temporary files that are created during compilation. Along with the `-v` option, this option is useful for debugging, especially when you have template functions or classes.

-KPIC

Same as the `-PIC` option, as described in “-PIC” on page 38.

-Kpic

Same as the -pic option, as described in "-pic" on page 38.

-L*dir*

Adds *dir* to the list of directories to be searched by the linker for libraries that contain object-library routines during the link step with ld. The directory, *dir*, is searched first.

Do not use this option or LD_LIBRARY_PATH to specify /usr/lib or /usr/ccs/lib, since these directories are searched by default.

You may see the following error message:

```
ld.so: library not found
```

while executing a program. It is displayed during the running of a.out, not during compilation or linking.

To correct the error, set LD_LIBRARY_PATH to include the directory where the missing library resides. Add the directory to the list of paths, rather than replacing the list of paths with the one directory. Here is a scenario:

- You are running OpenWindows™.
- You define the LD_LIBRARY_PATH environment variable to link in the XView libraries.
- While executing a program, you see the error message:
  `ld.so: library not found`

To fix this problem, set the LD_LIBRARY_PATH environment variable. In a Bourne Shell:

```
$ LD_LIBRARY_PATH=/opt/SUNWspro/lib :"$LD_LIBRARY_PATH"
$ export LD_LIBRARY_PATH
```

In a C shell:

```
% setenv LD_LIBRARY_PATH /opt/SUNWspro/lib:"$LD_LIBRARY_PATH":
```

The LD_LIBRARY_PATH has a list of directories, usually separated by colons. After you type a.out, the dynamic loader searches the directories in LD_LIBRARY_PATH before the default directories.

To see which libraries are linked dynamically in your executable, use the `ldd` command, as follows:

```
% ldd a.out
```

The *C++ 4.1 Library Reference Manual* and the *Tools.h++ Introduction and Reference Manual* contain further information on some of the C++ libraries.

-l*lib*

Specifies additional libraries for linking with object files. This option behaves like the `-l` option when it is used with `CC` or `ld`. Normal libraries have names like `lib`*something*`.a`, where the `lib` and `.a` parts are required. You can specify the *something* part with this option, and as many as you want on a single command-line; they are searched in order.

Use this option after your file name.

-libmieee

Causes `libm` to return values in the spirit of IEEE 754. The default behavior of `libm` is to be SVID-compliant on Solaris 2.*x*, and IEEE-compliant on Solaris 1.*x*.

-libmil

Inlines some library routines for faster execution. This option selects the best assembly language inline templates for the floating-point option and platform on your system.

Note – This option does not affect C++ inline functions.

-migration

Displays the contents of the *C++ Migration Guide*, which contains information about incompatibilities between C++ 3.0, based on `Cfront`, and the current compiler.

The contents of the file are paged through the command specified by the environment variable, `PAGER`. If this environment variable is not set, the default paging command is `more`.

-misalign

(SPARC) Informs the compiler that the data in your program is not properly aligned, such as that in the following code:

```
char b[100];
int f(int *ar){
return   *(int *)(b +2) +  *ar;
}
```

Thus, very conservative loads and stores must be used for the data, that is, one byte at a time. Using this option can cause significant degradation in performance when you run the program.

-mt

(Solaris 2.x) Compiles and links a multithreaded program, and passes `-D_REENTRANT` to the preprocessor. Passed to `ld` is the command:

```
-lsunmath_mt -lm -lC -lw -lthread -lcx -lc
```

instead of:

```
-lsunmath -lm -lC -lC_mtstubs -lw -lcx -lc
```

Use this option, rather than `-lthread`, to link with `libthread` to ensure proper library linking order.

-native

Chooses the correct code generator option. This option directs the compiler to generate code targeted for the machine that is doing the compilation.

Native floating point—use what is best for this machine.

This option is a synonym for `-xtarget=native`.

The `-fast` macro includes `-native` in its expansion.

-nocx*:*

(Solaris 1.x) (HP-UX) Makes the output executable smaller by not linking with `-lcx`. However, the runtime performance and accuracy of binary-decimal base conversion is somewhat compromised.

This option also disables the recognition of the long long type during compilation, and passes `-D_NO_LONGLONG` to the preprocessor. If one or more object files are compiled without `-nocx`, and they contain the use of long long, then you cannot pass `-nocx` to the compiler driver at link-time.

See also “-llib” on page 31.

-noex

Instructs the compiler to not generate code that supports C++ exceptions.

-nofstore

(x86) Does not convert the value of a floating point expression or function to the type on the left hand side of an assignment, when that expression or function is assigned to a variable or is cast to a shorter floating point type; rather, leaves the value in a register.

See also “-fstore” on page 25.

-nolib

Does not link with any system or library; that is, no `-l` options are passed to `ld`. You can then link some of the default libraries statically instead of dynamically.

You must pass all `-l` options explicitly. The `CC` driver normally passes the following options to `ld`:

Solaris 2.*x*	`-lsunmath -lm -lC -lC_mtstubs -lw -lcx -lc`
Solaris 1.*x*	`-lsunmath -lm -lC -lansi -lcx -lc`

On Solaris 2.*x*, link `libC`, `libw`, `lsunmath`, and `libm` statically, and `libc` dynamically as follows:

```
% CC test.c -nolib -Bstatic -lsunmath -lm -lC -lC_mtstubs -lw -lcx \
  -Bdynamic -lc
```

The order of the `-l` options is significant. The `-lsunmath`, `-lm`, `-lC`, `-lC_mtstubs`, `-lw`, and `-lcx` options must appear before `-lc`.

-nolibmil

Resets `-fast` so that it does *not* include inline templates. Use this option after the `-fast` option, as in:

```
% CC –fast –nolibmil
```

-noqueue

Returns without queuing your request and without compiling if no license is available. A nonzero status is returned for testing makefiles.

-norunpath

(Solaris 2.x) Does not build the path for shared libraries into the executable. If an executable file uses shared libraries, then the compiler normally builds in a path that points the runtime linker to those shared libraries. To do so, the compiler sets the environment variable, `LD_RUN_PATH`, before invoking `ld`. The path depends on the directory where you have installed the compiler.

This option is helpful if you have installed the compiler in some nonstandard location, and you ship an executable file to your customers, who need not work with that nonstandard location.

-O*level*

Optimizes the execution time. To optimize the execution time of `-O`, omit *level*. Doing so is equivalent to using level `-O2`.

There are four levels that you can use with `-O`. The following sections describe how they differ for SPARC systems, for x86, and for all systems.

For SPARC Systems

`-O1`: Does only the minimum amount of optimization (peephole), which is postpass assembly-level optimization. Do not use `-O1` unless using `-O2` or `-O3` results in excessive compilation time, or you are running out of swap space.

`-O2`: Does basic local and global optimization, which includes:

- Induction-variable elimination
- Local and global common-subexpression elimination
- Algebraic simplification
- Copy propagation
- Constant propagation
- Loop-invariant optimization
- Register allocation
- Basic block merging
- Tail recursion elimination
- Dead-code elimination
- Tail-call elimination
- Complex-expression expansion

This level does not optimize references or definitions for external or indirect variables. Do not use `–O2` unless using `–O3` results in excessive compilation time, or you are running out of swap space. In general, this level results in minimum code size.

`-O3:` Also optimizes references and definitions for external variables in addition to optimizations performed at the `-O2` level. This level does not trace the effects of pointer assignments. Do not use it when compiling either device drivers, or programs that modify external variables from within signal handlers. In general, this level results in increased code size.

`-O4:` Does automatic inlining of functions contained in the same file in addition to performing `-O3` optimizations. This automatic inlining usually improves execution speed, but sometimes makes it worse. In general, this level results in increased code size.

`-O5`: Generates the highest level of optimization. Uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time. Optimization at this level is more likely to improve performance if it is done with profile feedback. See -xprofile.

For x86, PowerPC and HP-UX Systems

`-O1`: Preloads arguments from memory; causes cross jumping (tail merging), as well as the single pass of the default optimization.

`-O2`: Schedules both high- and low-level instructions, and performs improved spill analysis, loop memory-reference elimination, register lifetime analysis, enhanced register allocation, global common subexpression elimination, as well as the optimization done by level 1.

`-O3`: Performs loop strength reduction, inlining, as well as the optimization done by level 2.

`-O4`: Performs architecture-specific optimization, as well as the optimization done by level 3.

For All Systems

For most programs:

- Level `–O5` is faster than `–O4`
- Level `–O4` is faster than `–O3`
- Level `-O3` is faster than `–O2`
- Level `-O2` is faster than `–O1`

In a few cases, `–O2` may perform better than the others, and `–O3` may outperform `–O4`. Try compiling with each level to see if you have one of these rare cases.

If the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization, and resumes subsequent procedures at the original level specified in the `–O` option.

If you optimize at `–O3` or `–O4` with very large procedures, such as thousands of lines of code in a single procedure, the optimizer may require an unreasonable amount of memory. In such cases, machine performance may be degraded.

To prevent this degradation from taking place, use the `limit` command to limit the amount of virtual memory available to a single process; see the `csh`(1) man page. For example, to limit virtual memory to 16 megabytes:

```
% limit datasize 16M
```

This command causes the optimizer to try to recover if it reaches 16 megabytes of data space.

This limit cannot be greater than the total available swap space of the machine, and should be small enough to permit normal use of the machine while a large compilation is in progress.

For example, on a machine with 32 megabytes of swap space, the command, `limit datasize 16M`, ensures that a single compilation never consumes more than half of the machine swap space.

The best setting of data size depends on the degree of optimization requested, the amount of real memory, and virtual memory available.

- To find the actual swap space, type: `swap -l`
- To find the actual real memory, type: `dmesg | grep mem`

`-o` *filename*

Sets the name of the output file (with the suffix, `.o`) or the executable file to *filename*. *filename* must have the appropriate suffix for the type of file to be produced by the compilation. It cannot be the same file as the source file, since the `CC` driver does not overwrite the source file.

`+p`

Disallows anachronistic constructs. The `p` in this option stands for “pure.”

The compiler usually warns you about anachronistic constructs. When you use this option, the compiler does not compile code containing anachronistic constructs.

See the *C++ Migration Guide* for more information.

`-P`

Runs a source file through the preprocessor, which outputs a file with a `.i` suffix. This option does not include preprocessor-type line number information in the output.

-p

(Not supported on HP-UX) Prepares the object code to collect data for profiling with `prof`. This option invokes a runtime recording mechanism that produces a `mon.out` file at normal termination.

You can also perform this task with the Analyzer. Refer to the `analyzer`(1) man page.

-pentium

(x86) Directs the compiler to generate code for the best performance on the Intel Pentium™ microprocessor.

See also “-native” on page 32.

-pg

(Not supported on HP-UX) Prepares the object code to collect data for profiling with `gprof`. This option invokes a runtime recording mechanism that produces a `gmon.out` file at normal termination.

You can also perform this task with the Analyzer. Refer to the `analyzer`(1) man page.

-PIC

Produces position-independent code.

(x86) Same as `-pic`.

(SPARC) (HP-UX) (PowerPC) Similar to `-pic`, but the global offset table spans the range of 32-bit addresses in those rare cases where there are too many global data objects for `-pic`.

-pic

Produces position-independent code. Use this option to compile source files when building a shared library.

Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table. The size of the global offset table is limited to 8 Kbytes on SPARC processors.

See also “-PIC” on page 38.

-pta

Directs the compiler to instantiate a whole template class, rather than only those functions which are used. This option creates a `.o` file for every member of a class. For it to work, you must reference at least one member of the class; otherwise, the compiler does not instantiate any members for the class.

-pti*path*

Includes the path name of the template source that you are using.

Use this option if you are not using the `-I` option. As it looks for template source path names, this option ignores path names provided with `-I`. By the same token, if you do not specify this option, the `-I` path names are used.

-pto

Instantiates generated templates in the compilation unit. Generated templates are then made static. The only restriction is that all template definitions must be located in the source or headers you are compiling.

-ptr*database-path*

Specifies the directory of primary and secondary build repositories. The template database is named `Templates.DB`; *database-path* is the directory containing the database, but does not include the name itself.

For example:

- `-ptr/tmp/Foo` specifies the database `/tmp/Foo/Templates.DB`
- `-ptr.` specifies the database, `./Templates.DB`

You can use multiple `-ptr` options. However, all repositories specified, except the first one, are read-only; no templates are instantiated into these directories.

If you omit this option, a default database, `./Templates.DB`, is created for you. To avoid confusion when multiple databases are involved, use `-ptr.` as the first entry on the command line.

-ptv

Turns on the verbose mode, sometimes called verify mode. The verbose mode displays each phase of instantiation as it occurs during compilation. Use this option if you are new to templates.

-Qoption|-qoption *prog opt*

Passes the option, *opt*, to the phase, *prog*. Refer to “Components” on page 13 for more information on compiler phases.

Table 2-1 shows the possible values for *prog*:

Table 2-1 Compiler Phases

SPARC Architecture	x86 and HP-UX Architectures
`ccfe`	`ccfe`
`iropt`	`cg386`
`cg`	`codegen`
`tdb_link`	`tdb_link`
`ld`	`ld`
`cdlink` *(Solaris 1.x only)*	

To pass more than one option, specify them in a specific order as a comma-separated list. In the following command-line, when `ld` is invoked by the `CC` driver, `-Qoption` passes the options, `-i` and `-m`, to `ld`:

```
% CC -Qoption ld -i,-m test.c
```

-qp

Same as the `-p` option, as described in “-p” on page 38.

-Qproduce|-qproduce *sourcetype*

Causes the CC driver to produce source code of the type *sourcetype*. Source code types are shown in Table 2-2.

Table 2-2 Source Code Types

.i	Preprocessed C++ source from ccfe
.o	Object file from cg, the code generator
.s	Assembler source from cg

-R:

(Solaris 1.x) Merges the data segment with the text segment. Data initialized in the object file produced by this compilation is read-only, and is shared between processes, unless linked with ld -N.

This option is ignored when -g is used.

-readme

Displays the contents of the README file, paged by the command specified by the environment variable, PAGER. If PAGER is not set, the default paging command is more.

For a description of the contents of this file, see “README file” on page xix.

-R *pathname*

(Solaris 2.x) Creates a colon-separated list of directories that are used to specify library search directories to the runtime linker. Multiple instances of -R*pathname* are concatenated, with each *pathname* separated by a colon. If both the LD_RUN_PATH and the -R option are specified, then the path from -R is scanned, and the path from LD_RUN_PATH is ignored.

This option is passed to ld.

-S

Causes the `CC` driver to compile the program and output an assembly source file, but not assemble the program. The assembly source file is named with a `.s` suffix.

-s

Removes all symbol information from output executable files. This option is passed to `ld`.

-sb

Causes the `CC` driver to generate extra symbol table information in a SourceBrowser database in the `.sb` directory for the `sbrowser` program.

-sbfast

Runs only the `ccfe` phase to generate an extra symbol table information in a SourceBrowser database in the `.sb` directory for the `sbrowser` program. No object file is generated.

-temp=*dir*

Sets the name of the directory for the temporary files, generated during the compilation process, to be *dir*.

-time

(*SPARC) (x86*) *(PowerPC)* Causes the `CC` driver to report execution times for various compilation passes.

-U*name*

Removes any initial definition of the macro symbol *name*. This option:

- Is processed by `ccfe`.
- Is the inverse of the `-D` option, as described in “-Dname[=def]” on page 20.

You can specify multiple `-U` options on the command-line.

`-unroll=`*n*

Specifies whether or not the compiler optimizes (unrolls) loops.

n is a positive integer; it works as follows:

- When *n* is `1`, it is a command, and the compiler unrolls no loops.
- When *n* is greater than `1`, `-unroll=`*n* causes the compiler to unroll loops *n* times.

`-V`

Directs the `CC` driver to print the names and version numbers of the programs it invokes.

`-v`

Prints the command-line for each compilation pass.

`+w`

Generates additional warnings about questionable constructs that are:

- Nonportable
- Likely to be mistakes
- Inefficient

By default, the compiler warns about constructs that are almost certainly problems.

`-w`

Causes the `CC` driver to *not* print warning messages from the compiler.

`-x601`

(PowerPC) Optimizes for the PowerPC 601 processor.

-x603

(PowerPC) Optimizes for the PowerPC 603 processor.

-x604

(PowerPC) Optimizes for the PowerPC 604 processor.

-xa

Same as the `-a` option, as described in “-a” on page 17.

-xar

Creates archive libraries.

When compiling a C++ archive that uses templates, it is necessary in most cases to include in the archive those template functions which are instantiated in the template database. Using this option automatically adds those templates to the archive as needed.

For example:

```
CC -xar -O libmain.a a.o b.o c.o
```

archives the template functions contained in the library and object files.

-xarch=*a*

(*Solaris 2.x) (SPARC*) Limits the set of instructions the compiler may use.

a must be one of: `generic`, `v7`, `v8a`, `v8`, `v8plus`, `v8plusa`.

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its *primary use* is to override a value supplied by the `-xtarget` option.

This option limits the instructions generated to those of the specified architecture, and *allows* the specified set of instructions. The option does not guarantee an instruction is used; however, under optimization, it is usually used.

If this option is used with optimization, the appropriate choice can provide good performance of the executable on the specified architecture. An inappropriate choice can result in serious degradation of performance.

`v7`, `v8a`, and `v8` are all binary compatible. `v8plus` and `v8plusa` are binary compatible with each other and forward, but not backward.

For any particular choice, the generated executable can run much more slowly on earlier architectures (to the left in the above list).

Table 2-3 The `-xarch` Values

Value	Meaning
`generic`	Gets good performance on most SPARCs, major degradation on none. This is the default. This option uses the best instruction set for good performance on most SPARC processors without major performance degradation on any of them. With each new release, this best instruction set will be adjusted, if appropriate.
`v7`	Limits instruction set to V7 architecture. This option uses the best instruction set for good performance on the V7 architecture, but without the quad-precision floating-point instructions. This is equivalent to using the best instruction set for good performance on the V8 architecture, but *without* the following instructions: The quad-precision floating-point instructions The integer `mul` and `div` instructions The `fsmuld` instruction Examples: SPARCstation 1, SPARCstation 2
`v8a`	Limits instruction set to the V8a version of the V8 architecture. This option uses the best instruction set for good performance on the V8 architecture, but without: The quad-precision floating-point instructions The `fsmuld` instruction Example: Any machine based on MicroSPARC I chip architecture

Table 2-3 The `-xarch` Values *(Continued)*

Value	Meaning
`v8`	Limits instruction set to V8 architecture. This option uses the best instruction set for good performance on the V8 architecture, but without quad-precision floating-point instructions. Example: SPARCstation 10
`v8plus`	Limits instruction set to the V8plus version of the V9 architecture. This option uses the best instruction set for good performance on the V9 architecture, but: Without the quad-precision floating-point instructions Limited to the 32-bit subset defined by the V8+ specification In V8+, a system with the 64-bit registers of V9 runs in 32-bit addressing mode, but the upper 32 bits of the i and l registers must not affect program results. This choice does not include the VIS instructions. Example: Any machine based on UltraSPARC chip architecture. Use of this option also causes the `.o` file to be marked as a V8+ binary. Such files will not run on a V8 machine.
`v8plusa`	Limits instruction set to the V8plusa version of the V9 architecture. This option uses the best instruction set for good performance on the UltraSPARC™ architecture, but limited to the 32-bit subset defined by the V8+ specification. This is equivalent to using the `v8plus` instruction set plus the UltraSPARC-specific instructions. This choice includes the VIS instructions. Example: Any machine based on UltraSPARC chip architecture Use of this option also causes the `.o` file to be marked as a Sun-specific V8+ binary. Such files will not run on a V8 machine.

-xcache=*c*

(*Solaris 2.x) (SPARC*) Defines the cache properties that the optimizercan use. It does not guarantee that any particular cache property is used.

c must be one of the following:

- `generic`
- *s1/l1/a1*
- *s1/l1/a1*:*s2/l2/a2*
- *s1/l1/a1*:*s2/l2/a2*:*s3/l3/a3*

That is, `-xcache=`{`generic` |*s1/l1/a1*[:*s2*/*l2*/*a2*[:*s3/l3/a3*]]}, where the *si/li/ai* are defined as follows:

si	The size of the data cache at level *i*, in kilobytes
li	The line size of the data cache at level *i*, in bytes
ai	The associativity of the data cache at level *i*

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its *primary use* is to override a value supplied by the `-xtarget` option.

.

Table 2-4 The `-xcache` Values

Value	Meaning
`generic`	Defines the cache properties for good performance on most SPARCs. This is the default value which directs the compiler to use cache properties for good performance on most SPARC processors, without major performance degradation on any of them.
s1/l1/a1	Defines level 1 cache properties.
s1/l1/a1:*s2/l2/a2*	Defines levels 1 and 2 cache properties.
s1/l1/a1:*s2/l2/a2*:*s3/l3/a3*	Defines levels 1, 2, and 3 cache properties

Example: `-xcache=16/32/4:1024/32/1` specifies the following:

Level 1 cache has:	Level 2 cache has:
16K bytes	1024K bytes
32 bytes line size	32 bytes line size
4-way associativity	Direct mapping associativity

`-xcg89`

(SPARC) Same as the -cg89 option. See *-cg[89|92]* on page 19.

`-xcg92`

(SPARC) Same as the -cg92 option. See *-cg[89|92]* on page 19.

`-xchip=`*c*

(*Solaris 2.x) (SPARC*) Specifies the target processor for use by the optimizer.

c must be one of: `generic`, `old`, `super`, `super2`, `micro`, `micro2`, `hyper`, `hyper2`, `powerup`, `ultra`

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its *primary use* is to override a value supplied by the `-xtarget` option.

This option specifies timing properties by specifying the target processor.

Some effects are:

- The ordering of instructions, that is, scheduling
- The way the compiler uses branches
- The instructions to use in cases where semantically equivalent alternatives are available

Table 2-5 The `-xchip` Values

Value	Meaning
`generic`	Uses timing properties for good performance on most SPARCs. This is the default value that directs the compiler to use the best timing properties for good performance on most SPARC processors, without major performance degradation on any of them.
`old`	Uses timing properties of pre-SuperSPARC™ processors.
`super`	Uses timing properties of the SuperSPARC chip.
`super2`	Uses timing properties of the SuperSPARC II chip.
`micro`	Uses timing properties of the MicroSPARC™ chip.
`micro2`	Uses timing properties of the MicroSPARC II chip.
`hyper`	Uses timing properties of the HyperSPARC™ chip.
`hyper2`	Uses timing properties of the HyperSPARC II chip.
`powerup`	Uses timing properties of the Weitek® PowerUp™ chip.
`ultra`	Uses timing properties of the UltraSPARC chip.

-xF

(Solaris 2.x) Enables reordering of functions, using the compiler, the Analyzer, and the linker.

If you compile with the `-xF` option, and then run the Analyzer, you generate a map file that shows an optimized order for the functions. The subsequent link to build the executable file can be directed to use that map by using the linker `-M`*mapfile* option.

If you include the `O` flag in the string of segment flags within the mapfile, then the static linker, `ld`, attempts to place sections in the order they appear in the mapfile, for example:

```
LOAD ? RXO
```

The `analyzer`(1), `debugger`(1), and `ld`(1) man pages provide further details on this option.

-xildoff

(*SPARC*) Turns off the incremental linker and forces the use of `ld`. This option is the default if you do *not* use the `-g` option. Override this default by using the `-xildon` option.

-xildon

(*SPARC*) Turns on the incremental linker and forces the use of `ild` in incremental mode. This option is the default if you use the `-g` option. Override this default by using the `-xildoff` option.

-xinline=*rlst*

Same as `-inline`, as described in “-inline=rlst” on page 28.

-xlibmieee

Same as `-libmieee`, as described in “-libmieee” on page 31.

-xlibmil

Same as `-libmil`, as described in “-libmil” on page 31.

-xlibmopt

(SPARC) Uses a math routine library optimized for performance. The results may be slightly different than those produced by the normal math library. This option is implied by the `-fast` option.

See also “-fast” on page 22 and “-xnolibmopt” on page 52.

-xlicinfo

Displays information on the licensing system. In particular, this option returns the license-server name and the user ID for each user who has a license checked out. When you use this option, the compiler is *not* invoked, and a license is *not* checked out.

-Xm

Allows the use of the character $ in identifier names, except as the first character.

-xM

Outputs makefile dependency information. For example, with the following code, `hello.c`:

```
#include <stdio.h>
main()
{
    (void) printf ("hello0");
}
```

When you compile with this option:

```
% CC -xM hello.c
```

The output shows the dependencies:

```
hello.o: hello.c
hello.o: /usr/include/stdio.h
```

See `make`(1) for details about makefiles and dependencies.

-xM1

Same as `-xM`, except that this option does not output dependencies for the `/usr/include` header files. When you compile the same code provided with the `-xM` option, the output is:

```
hello.o: hello.c
```

-xMerge

(Solaris 2.x) Merges the data segment with the text segment. The data in the object file is read-only, and is shared between processes, unless you link with `ld -N`.

-xnolib

Same as -nolib, as described in "-nolib" on page 33.

-xnolibmopt

(SPARC) Resets -fast, and does *not* use the math routine library.

Use this option *after* the -fast option on the command-line, as in:
CC -fast -xnolibmopt ....

-xO*level*

Same as -O*level*, as described in "-Olevel" on page 34.

-xpg

Same as -pg, as described in "-pg" on page 38.

-xprofile=*p*

(*Solaris 2.x*) (*SPARC*) Collects data for a profile or use a profile to optimize.

p must be collect, use[:*executable name*] or tcov. *Executable name* is the name of the executable that is being analyzed. The *executable name* is optional. If it is not specified, the *executable name* is assumed to be a.out.

This option causes execution frequency data to be collected and saved during the execution, then the data can be used in subsequent runs to improve performance.

The -xprofile=tcov and the -a options are compatible in a single executable. That is, you can link a program that contains some files which have been compiled with -xprofile=tcov, and others with -a. You cannot compile a single file with both options.

Table 2-6 The `-xprofile` Values

Value	Meaning
`collect`	Collects and saves execution frequency for later use by the optimizer. The compiler inserts code to measure the execution frequency at a low level. During execution, the measured frequency data is written into `.prof` files that correspond to each of the source files. If you run the program several times, the execution frequency data accumulates in the `.prof` files; that is, output from prior runs is not lost.
`use[:`*`en`*`]`	Uses execution frequency data saved by the compiler. Where en stands for *executable name*, the name of the executable that is being analyzed. The executable name is optional. If it is not specified, the executable name is assumed to be a.out. Optimizes by using the execution frequency data previously generated and saved in the `.prof` files by the compiler. The source files and the compiler options (excepting only this option), must be exactly the same as for the compilation used to create the compiled program that was executed to create the `.prof` files.
`tcov`	This option is the new style of basic block profiling for `tcov`. It has similar functionality to the `-a` option, but correctly collects data for programs that have source code in header files or make use of C++ templates. See `-a` for information on the old style of profiling, the `tcov`(1) man page, and the *Profiling Tools* manual for more details. Code instrumentation is performed similarly to that of the -a option, but .d files are no longer generated. Instead, a single file is generated, whose name is based off of the final executable. For example, if the program is run out of `/foo/bar/myprog`, then the data file is stored in `/foo/bar/myprog.profile/myprog.tcovd`. The `-xprofile=tcov` and the `-a` options are compatible in a single executable. That is, you can link a program that contains some files which have been compiled with `-xprofile=tcov`, and others with `-a`. You cannot compile a single file with both options. When running `tcov`, you must pass it the `-x` option to make it use the new style of data. If you don't, `tcov` uses the old `.d` files, if any, by default for data, and produces unexpected output. Unlike `-a`, the `TCOVDIR` environment variable has no effect at compile-time. However, its value is used at program runtime.

-xregs=*r*

(*Solaris 2.x*)(*SPARC*) Specifies the usage of registers for the generated code.

r is a comma-separated list that consists of one or more of the following: [no%]`appl`, [no%]`float`.

Example: `-xregs=appl,no%float`

Table 2-7 The `-xregs` Values

Value	Meaning
`appl`	Allows using the registers `g2`, `g3`, and `g4`. In the SPARC ABI, these registers are described as *application* registers. Using these registers can increase performance because fewer load and store instructions are needed. However, such use can conflict with some old library programs written in assembly code.
`no%appl`	Does not use the `appl` registers.
`float`	Allows using the floating-point registers as specified in the SPARC ABI. You can use these registers even if the program contains no floating-point code.
`no%float`	Does not use the floating-point registers. With this option, a source program cannot contain any floating-point code.

The default is `-xregs=appl,float`.

-xs

(Solaris 2.x) Disables Auto-Read for `dbx`. Use this option in case you cannot keep the `.o` files around. This option passes the `-s` option to the assembler.

No Auto-Read is the older way of loading symbol tables. It places all symbol tables for `dbx` in the executable file. The linker links more slowly, and `dbx` initializes more slowly.

If you move the executables to another directory, then to use `dbx`, you must move the source files, but you need not move the object (`.o`) files.

”Auto-Read is the newer and default way of loading symbol tables. With Auto-Read the information is placed in the `.o` files, so that `dbx` loads the information only if and when it is needed. Hence the linker links faster, and `dbx` initializes faster.

With `-xs`, if you move executables to another directory, then to use dbx, you can ignore the object (`.o`) files.

Without `-xs`, if you move the executables to another directory, to use `dbx`, you must move *both* the source files and the object (`.o`) files.

-xsafe=mem

(*Solaris 2.x*) (*SPARC*) Allows the compiler to assume no memory-based traps occur.

This option grants permission to use the speculative load instruction on V9 machines.

-xsb

Same as `-sb`, as described in “-sb” on page 42.

-xsbfast

Same as `-xsbfast`, as described in “-sbfast” on page 42.

–xspace

(*Solaris 2.x*) (*SPARC*) Does no optimizations that increase code size.

Example: Do not unroll loops.

-xtarget=*t*

(*Solaris 2.x*) (*SPARC*) Specifies the target system for instruction set and optimization.

t must be one of: `native`, `generic`, *system-name.* See Table 2-8 for the meanings of these `–xtarget` values.

The performance of some programs may benefit by providing the compiler with an accurate description of the target computer hardware. When program performance is critical, the proper specification of the target hardware could be very important. This is especially true when running on the newer SPARC processors. However, for most programs and older SPARC processors, the performance gain is negligible and a generic specification is sufficient.

Table 2-8 The `-xtarget` Values

Value	Meaning
`native`	Gets the best performance on the host system. The compiler generates code for the best performance on the host system. It determines the available architecture, chip, and cache properties of the machine on which the compiler is running.
`generic`	Gets the best performance for generic architecture, chip, and cache. The compiler expands `-xtarget=generic` to: `-xarch=generic -xchip=generic -xcache=generic` This is the default value.
system-name	Gets the best performance for the specified system. You select a system name from Table 2-9 that lists the mnemonic encodings of the actual system name and numbers.

The `–xtarget` option is a macro. Each specific value for `-xtarget` expands into a specific set of values for the `-xarch`, `-xchip`, and `-xcache` options. See Table 2-9 for the values. For example:

`-xtarget=sun4/15` is equivalent to:

```
-xarch=v8a -xchip=micro -xcache=2/16/1
```

Table 2-9 The `-xtarget` Expansions

-xtarget	-xarch	-xchip	-xcache
sun4/15	v8a	micro	2/16/1
sun4/20	v7	old	64/16/1
sun4/25	v7	old	64/32/1
sun4/30	v8a	micro	2/16/1
sun4/40	v7	old	64/16/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1
sun4/65	v7	old	64/16/1
sun4/75	v7	old	64/32/1
sun4/110	v7	old	2/16/1
sun4/150	v7	old	2/16/1
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/630	v7	old	64/32/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1

Table 2-9 The `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/64/1
ssslc	v7	old	64/16/1
ss1	v7	old	64/16/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1
ss2p	v7	powerup	64/32/1
ss4	v8a	micro2	8/64/1
ss5	v8a	micro2	8/64/1
ssvygr	v8a	micro2	8/64/1
ss10	v8	super	16/32/4
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/20	v8	super	16/32/4
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1

Table 2-9 The `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
ss10/402	v8	super	16/32/4
ss10/412	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1
ss10/514	v8	super	16/32/4:1024/32/1
ss10/612	v8	super	16/32/4:1024/32/1
ss10/712	v8	super2	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1
ss20/51	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1
ss20/502	v8	super	16/32/4
ss10/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/712	v8	super	16/32/4:1024/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/	v8	super	16/32/4:1024/32/1

Table 2-9 The `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
ss600/	v8	super	16/32/4:1024/32/1
ss600/	v8	super	16/32/4:1024/32/1
ss1000	v8	super	16/32/4:1024/32/1
sc2000	v8	super	16/32/4:1024/64/1
cs6400	v8	super	16/32/4:2048/64/1
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ultra	v8	ultra	16/32/1:512/64/1
ultra1/140	v8	ultra	16/32/1:512/64/1
ultra1/170	v8	ultra	16/32/1:512/64/1
ultra1/1170	v8	ultra	16/32/1:512/64/1
ultra1/2170	v8	ultra	16/32/1:512/64/1
ultra1/2200	v8	ultra	16/32/1:1024/64/1

-xtime

Same as `-time`, as described in "-time" on page 42.

-xunroll=*n*

Same as `-unroll=`*n*, as described in "-unroll=n" on page 43.

-xwe

Converts all warnings to errors by returning non-zero exit status.

-Ztha

(Solaris 2.x) (SPARC) Prepares code for analysis by the Thread Analyzer, the performance analysis tool for multithreaded code.

If you compile and link in separate steps, you should use this option in both steps. Code linked with this option is linked with the library, `libtha.so`.

`-ztext`

Forces fatal error if any relocations remain against non-writable, allocatable sections.

2

Templates 3

Before reading this chapter, you should be familiar with the *C++ Annotated Reference Manual* section on Templates, which will partially prepare you to use this implementation of templates. *We suggest you read this chapter in its entirety.*

How this C++ Implementation Differs from `Cfront`

This compiler's implementation of templates is different than that of AT&T's `Cfront` compiler.

`Cfront` *Link Time Instantiation*

Instantiation is the process by which a C++ compiler creates a usable function or object from a template. Two common methods of template instantiation are compile time instantiation and link time instantiation. `Cfront` uses the link time method, which uses this algorithm:

1. **Compile all user source files.**
2. **Using the prelinker, `ptlink`, link all object files created in step 1 into a partially linked executable.**
3. **Examine the link output and instantiate all undefined functions for which there are matching templates.**
4. **Link all created templates along with the partially linked executable files from step 2.**

5. **As long as there are undefined functions for which there are matching template functions, repeat steps 3 through 4.**
6. **Perform the final pass of the link phase on all created object files.**

Advantage of `Cfront` *Link Time Instantiation*

The main advantage of link time instantiation is that no special outside support is required to handle specializations, (user-provided functions intended to override instantiated template functions), since only those functions that have not been defined in the user source files become targets of instantiation by the compiler.

Disadvantages of `Cfront` *Link Time Instantiation*

The two main disadvantages of link time instantiation are:

- Error messages are deferred. Because an instantiation takes place during the link phase, all error messages resulting from an instantiation are deferred until *after* its use. As a result, there is no helpful traceback of where the error may be occurring.
- Repetitive calls to the prelinker can dramatically increase the link time.

SPARCompiler C++ Compile Time Instantiation

C++ 4.1 uses compile time instantiation, which forces instantiations to occur when the reference to the template is being compiled.

Advantages of Compile Time Instantiation

The advantages of compile time instantiation are:

- Debugging is much easier. Error messages occur within context, allowing the compiler to give a complete traceback to the point of reference.
- Template instantiations are always up-to-date.
- The overall compilation time, including the link phase, is reduced.

Disadvantages of Compile Time Instantiation

Templates may be instantiated multiple times if source files reside in different directories, or if you use libraries with template symbols.

Class Templates

Class templates provide a mechanism to create classes with parameterized types. For example, Code Example 3-1 creates an array class template:

Code Example 3-1 Array Class Template

```
const int ArraySize = 20;
template <class Type> class Array {
   private:
      Type* data;
      int size;
   public:
      Array(int sz=ArraySize);
      int GetSize();
};
template <class Type>        // constructor
Array<Type>::Array(int sz) {
   size = sz;
   data = new Type[ size ];
};
template <class Type>       // member function
int Array<Type>::GetSize()
{
   return size;
}
```

In this example, the class `Array` is declared as a template class, with a single replaceable type parameter `Type`. The class template is instantiated like this:

```
Array<int> IntArray(100);
```

The compiler instantiates an `Array` object that handles arrays of `int`s. In all locations where `Type` appears, `int` is substituted. When the compiler creates this object, it also creates the template constructor member function. The compiler does not instantiate a function unless it is explicitly referenced. For example, to create `GetSize()`, you reference it like this:

```
int x = IntArray.GetSize();
```

The compiler then instantiates the `GetSize()` template member function. With any class template, if referenced, a minimum of two template functions are instantiated: the constructor and destructor for the class template. Optionally, the compiler may generate any object files containing information about the class's virtual table and definitions for its static members.

Function Templates

Function templates specify how individual functions can be constructed. A family of sort functions could be declared like this:

```
template <class T> void sort (Array<T>);
```

and defined as:

```
template <class T> void sort ( Array<T> arrayToSort )
{
    // Do stuff with arrayToSort
}
```

In this example we declare a sort function that takes one of our predeclared `Array` class template objects. For this template to be actually instantiated, you must explicitly reference the function in question, such as:

```
Array<int> IntArray(100);   // Construct our array of ints
sort(IntArray);             // Sort it
```

Template Specializations

A template specialization is a specialized version of a class template member function that overrides the default instantiation if the default instantiation cannot handle a given type adequately. Code Example 3-2 shows a specialized constructor that creates an array of character strings of fixed size:

Code Example 3-2 Template Specialization

```
void Array<char*>::Array(int sz)
{
   size = sz;
   data = new char * [size];
   for (int i=0;i < size; i++) {
      data[i]=new char[128];
}
```

In this example the constructor allocates each `char` pointer in the array. Because a given executable may span many object files and libraries, the compiler does not know whether the specific function called exists somewhere else in the compilation environment. Specializations can be registered in the options file, as discussed in “The Options File” on page 76. Given the above source, the following takes place:

```
Array<int>   IntArray(10);      // The compiler instantiates an
                               // Array of ints using the default
                                 // constructor
Array<char*> CharStarArray(10); // The compiler uses the
                                 // specialized constructor
```

Code Example 3-3 illustrates how specializations can also be used for template functions for example, non template-class functions:

Code Example 3-3 Non-template Class Specialization

```
template <class T> void func ( T t ) { }   // A template function

void func(double) { }              // A specialization of func()

int main()
{
    func(10);    // The compiler-instantiated func<int> is used
    func(1.23);  // The user-provided func(double) is used
}
```

Organizing your Files

There are three methods by which you can organize your files and the templates within them. They are:

- Single file
- Combined file
- Multiple file

Single File Method

The single file method requires that the compiler see all the template definitions prior to their use. This method's limitation is that templates in one module are not usable in another module. Code Example 3-4 illustrates the single file method:

Code Example 3-4 Single File Method

main.cc

```
const int ArraySize = 20;
template <class Type> class Array {
    private:
        Type* data;
        int     size;
    public:
        Array(int sz=ArraySize);
        int GetSize();
};

template <class Type> Array<Type>::Array(int sz)
{
    size = sz;
    data = new Type[size];
}

template <class Type> int Array<Type>::GetSize()
{
    return size;
}

int main()
{
    Array<int> IntArray;
    int size = IntArray.GetSize();
    return size;
}
```

In this example, the compiler has seen all template declarations and definitions at the point of their use, and performs the required instantiations. For template functions, you can use the following:

main.cc	template <class T> void func (T t) { }
	
	int main()
	{
	    func(10);
	}

With template functions, the generated function `void func(int)` is accessible to other modules via a simple external declaration. However, doing so requires that the template function be forcibly instantiated in the main file, which may not happen as expected. Avoid using this function.

Combined File

The combined file method requires that the template class declarations and the member function definitions be established in a single `#include` file. This method allows for a template and its functions to be shared across multiple source files. Code Example 3-5 and Code Example 3-6 show the combined file method:

Code Example 3-5 Combined File Method – Header File

array.h

```
#ifndef _ARRAY_H_
#define _ARRAY_H_
const int ArraySize = 20;
template <class Type> class Array {  // template class declaration
    private:
        Type* data;
        int     size;
    public:
        Array(int sz=ArraySize);
        int GetSize();
};

// member function definitions

template <class Type> Array<Type>::Array(int sz)
{
    size = sz;
    data = new Type[size];
}

template <class Type> int Array<Type>::GetSize()
{
    return size;
}

#endif // _ARRAY_H_
```

Code Example 3-6 Combined File Method – Main Program

main.cc	#include "array.h"
	
	int main()
	{
	    Array<int> IntArray;
	    int size = IntArray.GetSize();
	    return size;
	}

With this method, templates are usable by more than one source module, a better solution than the single file method. For template functions, the method is the same; Code Example 3-7 illustrates the combined file method with template functions:

Code Example 3-7 Combined File Method with Template Functions

func.h	#ifndef _FUNC_H_ #define _FUNC_H_ template <class T> void func (T t) { } #endif // _FUNC_H_

main.cc	#include "func.h"
	
	int main()
	{
	    func(10);
	}

Because the declaration file `func.h` is included by any file using the template, the template's proper instantiation is guaranteed, removing the chance for errors present with the single file method.

Note – To use the `-pto` compile-line flag, you must use either the single file or combined file method.

Multiple File Method

The multiple file method calls for a header file where template classes are defined, and a source file where template member functions are defined. For example:

Code Example 3-8 Multiple File Method – Header File

array.h

```
#ifndef _ARRAY_H_
#define _ARRAY_H_
const int ArraySize = 20;
template <class Type> class Array {
    private:
        Type* data;
        int     size;
    public:
        Array(int sz=ArraySize);
        int GetSize();
};

#endif // _ARRAY_H_
```

Code Example 3-9 Multiple File Method – Source File

array.cc

```
#include "array.h"

template <class Type> Array<Type>::Array(int sz)
{
    size = sz;
    data = new Type[size];
}

template <class Type> int Array<Type>::GetSize()
{
    return size;
}
```

Code Example 3-10 Multiple File Method – Main Program

main.cc	#include "array.h"
	
	int main()
	{
	    Array<int> IntArray;
	    int size = IntArray.GetSize();
	    return size;
	}

Although the single file method is easier to read and understand, and is recommended for simple programs, the multiple file method gives you the greatest flexibility. Because of the separation of header and template source files, you must use greater care in file construction, placement, and naming.

The built-in preprocessor is an integral part of template compilation. Other versions of the C preprocessor may not be used unless a complete source of any template implementation files is included by that preprocessor. That is, separate template definition files are not supported exept by the built-in preprocessor.

For template functions, the system is identical in nature. For example:

Code Example 3-11 Multiple File Method – Header File

func.h	#ifndef _FUNC_H_ #define _FUNC_H_ template <class T> void func (T t); #endif // _FUNC_H_

Code Example 3-12 Multiple File Method – Source File

func.cc	#include "func.h"
	
	template <class T> void func (T t)
	{
	    // do stuff
	}

Code Example 3-13 Multiple File Method – Main Program

main.cc	#include "func.h"
	
	int main()
	{
	    func(10);
	}

Source File Location and Conventions

Without specific definitions as provided with an options file (see “The Options File” on page 76), the compiler uses a `Cfront`-style method to locate template definition files. This method requires that the template definition file contain the same base name as the template declaration file, and also be on the current `include` path. For example, if the template function `foo()` is located in `foo.h`, the matching template definition file should be named `foo.cc` or some other recognizable source-file extension. The template definition file must be located in one of the normal `include` directories or in the same directory as its matching header file.

The Template Database

The template database is a directory containing all configuration files needed to handle and instantiate the templates required by your program. It also acts as a repository for all generated object files containing templates.

The directory's name is `Templates.DB`. If the `Templates.DB` directory does not exist, and the compiler needs to instantiate a template, the directory is created for you.

Advantage: Auto Consistency

The template database manager ensures that the state of the files in the database is consistent and up-to-date with your source files.

For example, if your source files are compiled with `-g` (debugging on), the files you need from the database are also compiled with `-g`.

In addition, the template database tracks changes in your compilation. For example, if the first time you compile your sources, you have the `-DDEBUG` flag set to define the name `DEBUG`, the database tracks this. On a subsequent compile, if you omit this flag, the compiler reinstantiates those templates on which this dependency is set.

The Options File

The template options file, `Template.opt`, is a user-provided optional file that contains options needed for the compilation of templates along with the source location information. Primarily, this file allows for the registration of specializations needed by the program. If no options file is present, the compiler issues a message and creates a default options file.

An options file entry consists of a keyword followed by expected text terminated with a semi-colon (;). Entries can span multiple lines, although the keywords cannot be split. It is an ASCII file that has the following keywords and formats:

Comment Entries

```
# Comment
```

Definition Entries:

`definition` < *name* > `in` "*file-a*"[, "*file-b*" ... "*file-n*"]
[`nocheck` "*options*"] ;

The "*name*" entry informs the compiler where a template function body, static member initializer, or template member function's body can be found. Since the compiler knows what template it's looking for, a simple name, such as `foo,` is sufficient for template definition entries. Qualified names are *not* allowed or needed. Parentheses, return types or parameter lists are not allowed.

The definition entry is provided for those cases when the template declaration and definition file names do not follow the standard `Cfront`-style conventions. See "Source File Location and Conventions" on page 75.

The "*file-n*" entries specify in which file the template definition can be found. Multiple entries allow the compiler to search multiple files if the template name exists in more than one place. This method is required because the

options definition entry handles only simple names, and because C++ allows for function overloading. Only *one* definition entry per name is allowed. If `func` is defined in three files, then those three files *must* be listed in the definition entry line. Regardless of the return type or parameters, only the name itself counts.

The file names must be enclosed in quotes (" ").

The `nocheck` entry tells the compiler and template database manager to ignore certain options when checking dependencies. If you do not want the compiler to reinstantiate a template function because of the addition or deletion of a specific command-line flag, you should add that flag here.

The options themselves should be enclosed in quotes (" ").

These are some examples of definition entries:

Code Example 3-14 Normal Template Function

`foo.cc`	`template <class T> T foo( T t ) {}`
`Template.opt`	`definition foo in “foo.cc” nocheck "-g";`

In the above example the compiler locates the template function `foo` in `foo.cc`, and instantiates it. If a reinstantiation check were later required, the compiler ignores the `-g` option.

Code Example 3-15 Static Member Initializer

`foo.h`	`template <class T> class foo { static T* fooref; };`
`foo_statics.cc`	`#include “foo.h”` `template <class T> T* foo<T>::fooref = 0`
`Template.opt`	`definition fooref in “foo_statics.cc”;`

The name provided for the definition of `fooref` is a simple name and not a qualified name, such as `foo::fooref`. The reason for the definition entry is that the file name cannot be located using the default `Cfront`-style search rules. See “Source File Location and Conventions”.

Code Example 3-16 Template Member Function

`foo.h`	`template <class T> class foo { T* foofunc(T); };`
`foo_funcs.cc`	`#include “foo.h”` `template <class T> T* foo<T>::foofunc(T t) {}`
`Template.opt`	`definition foofunc in “foo_funcs.cc”;`

As the example above shows, template member functions are handled like the static member initializers.

Code Example 3-17 Multiple Files

`foo.h`	`template <class T> class foo {`
	`  T* func( T t );`
	`  T* func( T t, T x );`
	`};`

`foo1.cc`	`#include "foo.h"` `template <class T> T* foo<T>::func( T t ) { }`
`foo2.cc`	`#include "foo.h"` `template <class T> T* foo<T>::func( T t, T x ) { }`
`Template.opt`	`definition func in "foo1.cc", "foo2.cc";`

In the above example, the compiler must be able to find both definitions of the overloaded function `func()`. The definition entry tells the compiler where to find the appropriate function definitions.

Specialization Entries

`special` *declaration*;

This entry specifies to the compiler that this function is a specialization and should not be instantiated when encountered. When using the compile-time instantiation method, preregister specializations with an entry in the options file. The keyword is `special` and is followed by a legal C++-style declaration without return types. Overloading of the `special` entry is allowed. For example:

Code Example 3-18 Using the `special` Entry

`foo.h:`	`template <class T> T foo( T t ) {};`
`main.cc:`	`#include “foo.h”`
`Template.opt:`	`special foo(int);`

The above options file informs the compiler that the template function `foo()` should not be instantiated for the type `int`, and that a specialized version is provided by the user. Without the above entry in the options file, the function may be instantiated unnecessarily, resulting in errors:

Code Example 3-19 Multiple Instances of a Specialized Function

`foo.h`	`template <classT> T foo( T t ) { return t + t; }`

`file.cc`	`#include "foo.h"`
	
	`int func()`
	`{`
	`  return foo(10);`
	`}`

`main.cc`	`#include "foo.h"`
	
	`int foo( int i ) { return i * i; }`
	
	`int main()`
	`{`
	`  int x = foo(10);`
	`  int y = func();`
	`  return 0;`
	`}`

In the above example, when the compiler compiles `main.cc`, the specialized version of `foo` is correctly used, since the compiler has seen its definition. When `file.cc` is compiled, however, the compiler instantiates its own version of `foo`, since it doesn't know `foo` exists in `main.cc`. In most cases, this process results in a multiply-defined symbol during the link, but in some cases (especially libraries), the wrong function may be used, resulting in run time errors. If you use specialized versions of a function, you *should* register those specializations.

The `special` keyword can be overloaded, as in this example:

Code Example 3-20 Overloading the `special` Keyword

foo.h	`template <classT> T foo( T t ) {}`

main.cc	`#include "foo.h"`
	
	`int   foo( int   i ) {}`
	`char* foo( char* p ) {}`

Template.opt	`special foo(int);` `special foo(char*);`

To specialize an entire template class, use the `special` keyword:

Code Example 3-21 Specializing a Template Class

foo.h	`template <class T> class Foo {`
	`  ....`
	`  various members`
	`  ....`
	`};`

main.cc	`#include "foo.h"`
	`int main ()`
	`{`
	`  Foo<int> bar;`
	`  return 0;`
	`}`

Template.opt	`special class Foo;`

To specialize a specific class, follow Code Example 3-22

Code Example 3-22 Specializing a Specific Class

foo.h	template <class T> class Foo {
	....
	various members
	....
	};

main.cc	#include "foo.h"
	int main ()
	{
	Foo<int> bar;
	return 0;
	}

Template.opt	special class Foo<int>;

If a template class member is a static member, you must include the keyword `static` in your specialization entry:

Code Example 3-23 Specializing a Static Member

foo.h	template <class T> class Foo {
	public:
	static T func(T);
	};

main.cc	#include "foo.h"
	int main()
	{
	Foo<int> bar;
	return 0;
	}

The proper entry in the options file is

```
special static Foo<int>::func(int);
```

Instance Entries

There may be times when, because of an explicit external declaration of a function, the compiler does not know when to perform an instantiation. The `instance` entry handles this. Its format is:

`instance` *declaration*;

The *declaration* must be a legal C++-style declaration without return types. For example:

Code Example 3-24 Using the `instance` Entry

foo.cc	`template <class T> T foo( T t ) {}`

main.cc	`int foo( int );`
	
	`int main()`
	`{`
	`  return foo(10);`
	`}`

Templates.opt	`definition foo in "foo.cc";` `instance foo(int);`

In the above example, the compiler, not having seen a template declaration for `foo()` when processing `main.cc`, does not know whether `foo()` is a template function or an ordinary function. The options file instructs the compiler to instantiate `foo()` as a template function and points to the definition via the definition entry.

Multiple Options File Support

`include` "*options-file*";

During processing, the specified options file is textually included into the current options file. You can have more than one `include` statement and place them anywhere in the options file. They can also be nested.

Source File Extensions

You can specify different source file extensions for the compiler to search for when it is using its default `Cfront`-style source file locator mechanism. The format is:

```
extensions "ext-list";
```

The *ext-list* is a list of extensions for valid source files in a space-separated format such as:

```
extensions ".CC .c .cc .cpp";
```

In the absence of this entry in the options file, the valid extensions for which the compiler searches are: `.cc`, `.c`, `.cpp`, `.C` and `.cxx`.

Using the Same Template Database for Multiple Targets

If you use the same template database for multiple targets, inconsistent results can occur. However, if you choose to build multiple targets in the same work directory, `tdb_link` notes this fact during parallel builds and implements a locking mechanism on the primary working database. If two or more links are attempted on the same database, `tdb_link` initiates a wait until the lock is cleared by the other compilation.

The `ptclean` *Command*

Changes in your program can render some instantiations superfluous. The `ptclean` command clears out the template database to ensure all instantiations are up-to-date. `ptclean` removes all instantiations, temporary files, and dependency files from the database.

Command-Line Options

These are the template-related command-line options:

`-pta`

This `Cfront`-style flag for backward compatibility causes the compiler to instantiate a whole template class rather than only those functions that are used. This instantiation creates one `.o` file for each member of a class. For the `-pta` option to work, you must reference at least one member of the class, otherwise the compiler does not instantiate any members for the class.

`-pti`*path*

This flag allows for the specification of the template source file `include` path if the `-I` flag is not used. You should use the `-I` flag (`include` directive) instead of this flag. If `-pti` is used, the template system looks for template definition files on these paths and ignores the `-I` paths. You can insert multiple `-pti` directives on the command-line.

`-pto`

This flag is for single-file mode compilation. All generated output goes into a single object file. All definitions for templates used must be seen prior to their use. This method does not use any repositories, and its primary purpose is speed of compilation and compatibility.

`-ptr`*database-path*

This optional flag can be used to specify a different default working database or even multiple databases. If you use multiple databases, the first one you specify is your working, writeable database. All others are read-only. To avoid confusion in multiple database environments, make the first entry your current working directory (as in `"-ptr."`).

`-pts`

This is a Cfront-style flag that in the C++ 4.1 compiler is set to behave identically to the `-pta` flag. See “`-pta`” for further information.

`-ptv`

This flag causes verbose output. It will generate messages regarding template instantiations that are taking place.

Potential Problem Areas

This section describes potential problem areas in using templates.

Local Types as Template Arguments

The template instantiation system relies on type-name equivalence to determine which templates need to be instantiated or reinstantiated. Thus local types can cause serious problems when used as template arguments. Beware of creating similar problems in your code. For example:

Code Example 3-25 Local Types as Template Arguments

array.h

```
#ifndef _ARRAY_H_
#define _ARRAY_H_
const int ArraySize = 20;
template <class Type> class Array {
    private:
        Type* data;
        int   size;
    public:
        Array(int sz=ArraySize);
        int GetSize();
};

#endif // _ARRAY_H_
```

array.cc

```
#include "array.h"

template <class Type> Array<Type>::Array(int sz)
{
    size = sz;
    data = new Type[size];
}

template <class Type> int Array<Type>::GetSize()
{
    return size;
}
```

file1.cc	#include "array.h"
	
	struct Foo {
	    int data;
	};
	
	Array<Foo> File1Data;

file2.cc	#include "array.h"
	
	struct Foo {
	    double data;
	};
	
	Array<Foo> File2Data;

The `Foo` type as registered in `file1.cc` is not the same as the `Foo` type registered in `file2.cc`. Using local types in this way could lead to errors and unexpected results.

Declarations of Template Functions

The template instantiation system requires that a declaration must be seen to follow the `Cfront`-style search rules in locating template definitions. This must be a true declaration, and not a `friend` declaration. For example:

Code Example 3-26 Declarations of Template Functions

array.h

```
#ifndef _ARRAY_H
#define _ARRAY_H
#include <iostream.h>

template <class T> class array {
    protected:
        int size;
    public:
        array();
        friend ostream& operator<<(ostream&, const array<T>&);
};

#endif // _ARRAY_H
```

array.cc

```
#include <iostream.h>
#include <stdlib.h>
#include "array.h"

template <class T> array<T>::array()
{
  size=1024;
}

template <class T> ostream& operator<<(ostream& out, const
                      array<T>& rhs)
{
    out << "[" << rhs.size << "]";
    return out;
}
```

main.cc	#include <iostream.h>
	#include "array.h"
	
	int main()
	{
	    cout << "creating an array of int... " << flush;
	    array<int> foo;
	    cout << "done\n";
	    cout << foo << endl;
	    return 0;
	}

When the compilation system attempts to link the produced object files, it will generate an undefined error for the `operator<<` function, and is *not* instantiated. There is not an error message during compilation because the compiler will have read:

```
friend ostream& operator<<(ostream&, const array<T>&);
```

as the declaration of a normal function, which is a `friend` of the `array` class. To properly instantiate the functions, the following declaration must be added outside the array class declaration:

```
template <class T> ostream& operator<<(ostream&, const
                      array<T>&);
```

This declaration guarantees that the function is instantiated for each object of type `array<T>`.

Building Archives with Templates

For information on building archives that use templates, see “-xar” on page 44.

Exception Handling 4

This chapter identifies exception handling as currently implemented in the C++ compiler. It also identifies some areas of exception handling that are not clearly defined in the ANSI X3J16 draft working paper. Some of the topics covered in this chapter are interpretations of the draft. Other topics are specific to this compiler implementation.

For additional information on exception handling, see *The C++ Programming Language (*Second Edition*)* by Bjarne Stroustrup.

Why Exception Handling?

Exceptions are anomalies that occur during the normal flow of a program and prevent it from continuing. These anomalies—user, logic, or system errors—can be detected by a function. If the detecting function cannot deal with the anomaly, it raises, or throws, an exception. A function that handles that kind of exception, catches it. This function is also known as an exception handler.

In C++, whenever an exception is thrown, it cannot be ignored—there must be some kind of notification or termination of the program. If no user-provided exception handler is present, the compiler provides a default mechanism to terminate the program.

The significant benefit of using exceptions is that it significantly reduces the size and complexity of program code and eliminates the need to explicitly test for specific program anomalies.

4

Using Exception Handling

There are three keywords for exception handling in C++:

- `try`
- `catch`
- `throw`

`try`

A `try` block is a group of C++ statements, normally enclosed in braces `{ }`, which may cause an exception. This grouping restricts exception handlers to exceptions generated within the `try` block.

`catch`

A `catch` block is a group of C++ statements that are used to handle a specific raised exception. `Catch` blocks, or handlers, should be placed after each `try` block. A `catch` block is specified by:

- The keyword `catch`
- A `catch` expression, which corresponds to a specific type of exception that may be thrown by the `try` block
- A group of statements, enclosed in braces `{ }` whose purpose is to handle the exception

`throw`

The `throw` statement is used to throw an exception to a subsequent exception handler. A `throw` statement is specified with:

- The keyword `throw`
- An assignment expression. The type of the result of the expression determines which `catch` exception handler receives control

An Example

Code Example 4-1 Exception Handling Example

```
class Overflow {
   // ...
public:
   Overflow(char,double,double);
};

void f(double x)
{
   // ...
   throw Overflow('+',x,3.45e107);
}

int main() {
    try {
        // ...
        f(1.2);
        //...
    }
    catch(Overflow& oo) {
        // handle exceptions of type Overflow here
    }
}
```

In this example, the function call in the `try` block passes control to `f()`, which throws an exception of type `Overflow`. This exception is handled by the `catch` block, which handles type `Overflow` exceptions.

Implementing Exception Handlers

The basic rules for implementing exception handlers are:

- When a function is called by many other functions, you should code it so an exception is thrown whenever an error is detected. Usually, the `throw` expression throws an object. This object is used to identify the types of exceptions and to pass specific information about the exception that has been thrown.

- Use the `try` statement in a client program to anticipate exceptions. Precede function calls that you anticipate may produce an exception with the keyword `try` and enclose the calls in braces.
- Code one or more `catch` blocks immediately after the `try` block. Each `catch` block identifies what type or class of objects it is capable of catching. When an object is thrown by the exception, this is what takes place:
  - If the object thrown by the exception matches the type of `catch` expression, control passes to that `catch` block.
  - If the object thrown by the exception does not match the first `catch` block, subsequent `catch` blocks are searched for a matching type.
  - If `try` blocks are nested, and there is no match, control passes from the innermost `catch` block to the outermost `catch` block.
  - If there is no match in any of the `catch` blocks, the program is normally terminated with a call to the predefined function `terminate()`. By default, `terminate()` calls `abort()`, which destroys all remaining objects and exits from the program. This default behavior can be changed by calling the `set_terminate()` function.

Synchronous Exception Handling

Exception handling is designed to support only *synchronous exceptions* such as array range checks. The term synchronous exception means that exceptions can only be originated from `throw` expressions.

The current draft supports synchronous exception handling with termination model. Termination means that once an exception is thrown, control never returns to the throw point.

Asynchronous Exception Handling

Exception handling is not designed to directly handle asynchronous exceptions such as keyboard interrupts. However, exception handling can be made to work in the presence of asynchronous events if care is taken. For instance, to make exception handling work with signals, you can write a signal handler that sets a global variable, have another routine that polls the value of that variable at regular intervals, and throws an exception when the value changes.

Flow of Control

In C++, exception handlers do not correct the exception and then return to the point at which the exception occurred. Instead, when an exception is generated, control is passed out of the function that threw the exception, out of the `try` block that anticipated the exception, and into the `catch` block whose exception declaration matches the exception thrown.

The `catch` block handles the exception. It either rethrows the exception, branches to a label, or ends normally. If a `catch` block ends normally, without a `throw`, the flow of control passes over all subsequent `catch` blocks.

Whenever an exception is thrown, caught, and control returned outside of the function that threw the exception, stack unwinding takes place. During stack unwinding, any automatic objects which were created within the scope of that function are safely destroyed via calls to their destructors.

If a `try` block ends without an exception, all subsequent `catch` blocks are ignored.

Note – An exception handler cannot return control to the source of the error by using the `return` statement. A `return` issued in this context returns from the function containing the `catch` block.

Branching Into and Out of `try` Blocks and Handlers

Branching out of a `try` block or a handler is allowed. Branching into a `try` block is allowed, but not advisable, since it makes the program not portable. Branching into a `catch` block is not allowed, however, because that is equivalent to jumping past an initiation of the exception.

Nesting of Exceptions

Exceptions can be nested. Nesting of exceptions occurs when exceptions are thrown in a handler, since the handled exception needs to be kept around because it can be rethrown, or when exceptions are thrown in the destructor during stack unwinding.

Using `throw` *in a Function Declaration*

A function declaration can include an exception specification, a list of exceptions that a function may directly or indirectly throw.

This declaration indicates to the caller that the function `foo` generates only one exception, and that it is caught by a handler of type `X`:

```
void foo(int) throw(X);
```

A variation on the previous example is:

```
void foo(int) throw();
```

This declaration guarantees that no exception is generated by the function `foo`. If an exception occurs, it results in a call to the predefined function `unexpected()`. By default, `unexpected()` calls `abort()` to exit the program. This default behavior can be changed by calling the `set_unexpected()` function; see “set_terminate() and set_unexpected() Functions” on page 95.

The check for unexpected exceptions is done at program execution time, not at compile time. The compiler may, however, eliminate unnecessary checking in some simple cases.

For instance, no checking for `f` is generated in the following example:

```
void foo(int) throw(x);
void f(int) throw(x);
{
    foo(13);
}
```

The absence of an exception specification allows any exception to be thrown.

Run-Time Errors

There are five run time error messages associated with exceptions. They are:

`No handler for the exception.`

```
Unexpected exception thrown.

An exception can only be re-thrown in a handler.

During stack unwinding, a destructor must handle its own
exception.

Out of memory.
```

When errors are detected at runtime the error message displays the type of the current exception and one of the above messages. The predefined function `terminate()` is then called, which calls `abort()` by default.

Caution – Selecting a `terminate()` function that does not terminate is an error.

The class `xunexpected` is now defined in `exception.h`.

The compiler makes use of the information provided in the exception specification in optimizing code production. For instance, table entries for functions that do not throw exceptions are suppressed, and run-time checkings for exception specifications of functions are eliminated wherever possible. Thus, declaring functions with correct exception specifications can lead to better code generation.

`set_terminate()` *and* `set_unexpected()` *Functions*

The following sections describe how to modify the behavior of the `terminate()` and `unexpected()` functions using `set_terminate()` and `set_unexpected()`.

`set_terminate()`

You can modify the default behavior of `terminate()` by calling the function `set_terminate()`:

```
typedef void (*PFV)();
PFV set_terminate(PFV);
```

`terminate()` calls the function passed as an argument to `set_terminate()`. The function passed in the most recent call to `set_terminate()` is called. The previous function passed as an argument to `set_terminate()` is the return value, so you can implement a stack strategy for using `terminate()`.

`set_unexpected()`

You can modify the default behavior of `unexpected()` by calling the function `set_unexpected()`:

```
typedef void (*PFV)()
PFV set_unexpected(PFV);
```

`unexpected()` calls the function passed as an argument to `set_unexpected()`. The function passed in the most recent call to `set_unexpected()` is called. The previous function passed as an argument to `set_unexpected()` is the return value; so you can implement a stack strategy for using `unexpected()`.

Matching Exceptions with Handlers

A handler type `T` matches a `throw` type `E` if any of the following is true:

1. `T` is same as `E`;
2. `T` is `const` or `volatile` of `E`;
3. `E` is `const` or `volatile` of `T`;
4. `T` is ref of `E` or `E` is ref of `T`;
5. `T` is a `public` or `protected` base of `E`;
6. `T` and `E` are both pointer types, and `E` can be converted to `T` by standard pointer conversion.

Throwing exceptions of reference or pointer types can result in a dangling pointer.

While handlers of type (X) and (X&) both match an exception of type X, the semantics are different. Using a handler with type (X) invokes the object's copy constructor and possibly truncates the object, which can happen when the exception is derived from X.

Handlers for a `try` block are tried in the order of their appearance. Handlers for a derived class (or a pointer to a reference to a derived class) must precede handlers for the base class to ensure that the handler for the derived class is invoked.

Access Control in Exceptions

The compiler performs the following check on access control on exceptions:

1. The formal argument of a `catch` clause obeys the same rules as an argument of the function in which the `catch` clause occurs.
2. An object can be thrown if it can be copied and destroyed in the context of the function in which the `throw` occurs.

Currently, access controls do not affect matching.

No other access is checked at runtime except for the matching rule described in "Matching Exceptions with Handlers" on page 96.

`-noex` *Compiler Option*

If you know that exceptions are not used, use the compiler option `-noex` to suppress generation of code that supports exception handling. The use of `-noex` results in smaller code size and faster code execution. When files compiled with `-noex` are linked to files compiled without `-noex`, some local objects are not destroyed when exceptions occur. By default, the compiler generates code to support exception handling.

New Runtime Function and Predefined Exceptions

You can use several functions related to exception handling. The header file `exception.h` includes the predefined exceptions `xmsg` and `xalloc`. The definitions provided in `exception.h` differ from those in the X3J16 Draft Working Paper.

4

Default `new-handler()` *Function*

When `::operator new()` cannot allocate storage, it calls the currently installed *new-handler* function. The default *new-handler* function throws an `xalloc` exception.

Note – The old behavior was to return a null from `::operator new()` when a memory request cannot be satisfied. To restore the old behavior, call `set_new_handler(0)`.

Building Shared Libraries with Exceptions

(Solaris 1.x) When building a shared library, exceptions involving functions from the library may not work unless the library is built with the `-G` option. See the *C++ 4.1 Library Reference Manual* for more on building and using shared libraries.

Note – When shared libraries are opened with `dlopen, RTLD_GLOBAL` must be used for exceptions to work.

(HP-UX) Exceptions are not supported in shared libraries built by C++ 4.1.

Using Exceptions in a Multithreaded Environment

The current exception-handling implementation is mt-safe—exceptions in one thread do not interfere with exceptions in other threads. However, you cannot use exceptions to communicate across threads; an exception thrown from one thread cannot be caught in another.

Each thread can set its own `terminate()` or `unexpected()` function. Calling `set_terminate()` or `set_unexpected()` in one thread affects only the exceptions in that thread. The default function for `terminate()` is `abort()` for the main thread, and `thr_exit()` for other threads. See “set_terminate() and set_unexpected() Functions” on page 95.

Using C and C++ 5

This chapter describes how to move programs from C to C++, and areas where C programs may have to be changed.

C programs generally require little modification to compile as C++ programs. C and C++ are link compatible. You don't have to modify compiled C code to link it with C++ code.

See *The C++ Programming Language*, by Bjarne Stroustrup, for more specific information on the C++ language.

5

Reserved and Predefined Words

Table 5-1 shows all reserved keywords in C++ and C, plus keywords that are predefined by C++. Keywords that are reserved in C++ and not in C are shown in **boldface.**

Table 5-1 Reserved Keywords

`__STDC__`	`default`	`if`	`short`	`union`
`__cplusplus`	**`delete`**	**`inline`**	`signed`	`unsigned`
`asm`	`do`	`int`	`sizeof`	**`virtual`**
`auto`	`double`	`long`	`static`	`void`
`break`	`else`	**`new`**	`struct`	`volatile`
`case`	`enum`	**`operator`**	`switch`	**`wchar_t`**
`catch`	`extern`	**`private`**	**`template`**	`while`
`char`	`float`	**`protected`**	**`this`**	
`class`	`for`	**`public`**	**`throw`**	
`const`	**`friend`**	`register`	**`try`**	
`continue`	`goto`	`return`	`typedef`	

`__STDC__` is predefined to the value 0. For example, Code Example 5-1:

Code Example 5-1 Predefines

```
#include <stdio.h>
main()
{
    #ifdef __STDC__
        printf("yes\n");
    #else
        printf("no\n");
    #endif

    #if    __STDC_ _ ==0
        printf("yes\n");
    #else
        printf("no\n");
    #endif
}
```

produces:

```
yes
yes
```

The following table lists reserved words for alternate representations of certain operators and punctuators specified in the current ANSI/ISO working paper from the ISO C++ Standards Committee. These alternate representations have not yet been implemented in the C++ compiler, but in future releases may be adopted as reserved words and should not be otherwise used.

Table 5-2 Reserved Words for Operators and Punctuators

`and`	`bitor`	`not`	`or`	`xor`
`and_eq`	`compl`	`not_eq`	`or_eq`	`xor_eq`
`bitand`				

Data Types

The basic types and their sizes are:

- `char` (1 byte)
- `short int` (2 bytes)
- `int` (4 bytes)
- `long int` (4 bytes)
- `long long int`[1] (8 bytes)
- wchar_t (4 bytes on Solaris 2.*x*, 2 on Solaris 1.*x*)
- enum (4 bytes)

Each of `char`, `short`, `int`, `long`, and `long long` can be prefixed with `signed` or `unsigned`. A type specified with `signed` is the same as the type specified without `signed`, except for `char` and bitfields.

- `float` (4 bytes)
- `double` (8 bytes)
- `long double` (12 bytes on Intel, 16 otherwise)
- `void`

1. `long long` is not available in `-Xc` mode, and in `-nocx` mode (Solaris 1.*x*).

5

Creating Generic Header Files

K&R C, ANSI C, and C++ require different header files. To make C++ header files conform to K&R C and ANSI C standards so that they are generic, use the macro `__cplusplus` to separate C++ code from C code. The macro `__STDC__` is defined in both ANSI C and C++. Use this macro to separate C++ or ANSI C code from K&R C code.

The `__cplusplus` *Macro*

You can insert an `#ifdef` statement in your code to conditionally compile C++ or C using the C++ compiler. To do this, use the `__cplusplus` macro:

```
#ifdef __cplusplus
int printf(char*,...);// C++ declaration
#else
int printf();/* C declaration */
#endif
```

Note – In the past, this macro was `c_plusplus`, which is no longer accepted.

Linking to C Functions

The compiler encodes C++ function names to allow overloading. To call a C function or a C++ function “masquerading” as a C function, you must prevent this encoding. Do so by using the `extern "C"` declaration. For example:

```
extern "C" {
double sqrt(double); //sqrt(double) has C linkage
    }
```

This linkage specification does not affect the semantics of the program using `sqrt()`, but simply causes the compiler to use the C naming conventions for `sqrt()`.

Only one instance of an overloaded C++ function can have C linkage. You can use C linkage for C++ functions that you intend to call from a C program, but you would only be able to use one instance of that function.

You cannot specify C linkage inside a function definition. Such declarations can only be done at the global scope.

5

FORTRAN Interface 6

This chapter describes the FORTRAN interface with C++. We suggest you follow these steps:

1. **Study Code Example 6-1 and “Sample Interface” on page 106.**
2. **Read “FORTRAN Calls C++” on page 114 or “C++ Calls FORTRAN” on page 138.**
3. **Within any of the two sections mentioned in step 2, choose one of these subsections:**
   - Arguments Passed by Reference
   - Arguments Passed by Value
   - Function Return Values
   - Labeled Common
   - Sharing I/O
   - Alternate Returns
4. **Within any of the subsections mentioned in step 3, choose one of these examples:**

   For the arguments, there is an example for each of these:
   - Simple Types (`character*1`, `logical`, `integer`, `real`, `double precision`)
   - Complex Types (`complex`, `double complex`)
   - Character Strings (`character*n`)
   - One-Dimensional Arrays (`integer a(9)`)
   - Two-Dimensional Arrays (`integer a(4,4)`)

- Structured Records (`structure` & `record`)
- Pointers

For the function return values, there is an example for each of these:

- Integer (`int`)
- Real (`float`)
- Pointer to real (pointer to `float`)
- Double precision (`double`)
- Complex
- Character string

Sample Interface

In Code Example 6-1, a FORTRAN main calls a C++ function. Both `i` and `f` are references.

Code Example 6-1 Sample C++ — FORTRAN Interface

Samp.cc	extern "C" void samp (int &i, float& f) {
	    i = 9;
	    f = 9.9;
	}

Sampmain.f	integer i real r external Samp !$pragma C (Samp) call Samp (i, r) write(*, "(I2, F4.1)") i, r end

Here, both `i` and `r` are passed by reference to the default. The following command-lines compile and execute `Samp.cc`, with output:

```
% CC -c Samp.cc
% f77 -c -silent Sampmain.f
% CC Samp.o Sampmain.o -lF77 -lM77
% a.out
 9 9.9
```

Compatibility Issues

Most C++ and FORTRAN interfaces must be consistent in the following:

- Function and Subroutine: definition and call
- Data types: compatibility of types
- Arguments: pass by reference or value
- Procedure name: uppercase and lowercase and trailing underscore (_)
- Libraries: Use FORTRAN libraries to link

Some C++ and FORTRAN interfaces must also get these right:

- Arrays: indexing and order
- File descriptors and `stdio`
- File permissions

Function versus Subroutine

The word *function* means different things in C++ and FORTRAN.

- As far as C++ is concerned, all subroutines are functions; the difference is that some of them return a null value.
- As far as FORTRAN is concerned, a function passes a return value; a subroutine does not.

FORTRAN Calls C++

If the C++ function returns a value, call it from FORTRAN as a function. If it does not return a value, call it as a subroutine.

C++ Calls FORTRAN

If it is a FORTRAN function, call it from C++ as a function. If it is a FORTRAN subroutine, call it from C++ as a function that returns a value of `int` (comparable to FORTRAN `INTEGER*4`) or `void`. This return value is useful if the FORTRAN routine does a nonstandard return.

6

Data Type Compatibility

Table 6-1 shows the default data type sizes and alignments (that is, without `-f`, `-i2`, `-misalign`, `-r4`, or `-r8`).

Table 6-1 Argument Sizes and Alignments, Pass by Reference

FORTRAN Type	C++ Type	Size (bytes)	Alignment (bytes)
byte x	char x	1	1
character x	char x	1	1
character*_n_ x	char x[n]	n	1
complex x	struct {float r,i;} x;	8	4
complex*8 x	struct {float r,i;} x;	8	4
double complex x	struct {double dr,di;}x;	16	4
complex*16 x	struct {double dr,di;}x;	16	4
double precision x	double x	8	4
real x	float x	4	4
real*4 x	float x	4	4
real*8 x	double x	8	4
integer x	int x	4	4
integer*2 x	short x	2	2
integer*4 x	int x	4	4
logical x	int x	4	4
logical*4 x	int x	4	4
logical*2 x	short x	2	2
logical*1 x	char x	1	1

The REAL*16 and th COMPLEX*32 can be passed between FORTRAN and C++, but not between FORTRAN and some previous versions C++.

Note that:

- Alignments are for FORTRAN types.
- Arrays pass by reference, if the elements are compatible.

- Structures pass by reference, if the fields are compatible.
- When passing arguments by value:
  - You cannot pass arrays, character strings, or structures by value.
  - You can pass arguments by value from FORTRAN to C++, but not from C++ to FORTRAN, since the `%VAL()` does not work in a `SUBROUTINE` statement.

Arguments Passed by Reference or Value

In general, FORTRAN passes arguments by reference. In a call, if you enclose an argument with the nonstandard function `%VAL()`, FORTRAN passes it by value.

In C++, the function declaration tells whether an argument is passed by value or by reference.

Uppercase and Lowercase

C++ is case-sensitive, uppercase and lowercase have different meanings. The FORTRAN default is to ignore case by converting subprogram names to lowercase, except within character-string constants.

There are two common solutions to the uppercase and lowercase problem:

- In the C++ subprogram, make the name of the C++ function all lowercase in order to match FORTRAN default behavior.
- Compile the FORTRAN program with the `-U` option. FORTRAN then preserves the existing uppercase and lowercase distinctions, and does not convert to all lowercase letters.

Use one of these solutions, but not both.

Most examples in this chapter use all lowercase letters for the name in the C++ function, and do not use the FORTRAN `–U` compiler option.

Underscore in Names of Routines

The FORTRAN compiler normally appends an underscore (_) to the names of subprograms, for both a subprogram and a call to a subprogram. The underscore distinguishes it from C++ procedures or external variables with the

same user-assigned name. If the name has exactly 32 characters, the underscore is not appended. All FORTRAN library procedure names have double leading underscores to reduce clashes with user-assigned subroutine names.

There are two common solutions to the underscore problem:

- In the C++ function, change the name of the function by appending an underscore to that name.
- Use the `C()` pragma to instruct the FORTRAN compiler to omit those trailing underscores.

Use one of these solutions, but not both.

Most of the examples in this chapter use the FORTRAN `C()` compiler pragma and do not use the underscores.

The `C()` pragma directive takes the names of external functions as arguments. It specifies that these functions are written in the C or C++ language, so the FORTRAN compiler does not append an underscore to such names, as it ordinarily does with external names. The `C()` directive for a particular function must appear before the first reference to that function. It must also appear in each subprogram that contains such a reference. The conventional usage is:

```
    EXTERNAL ABC, XYZ!$PRAGMA C( ABC, XYZ )
```

If you use this pragma, then in the C++ function do not append an underscore to those names.

C++ Name Encoding

To implement function overloading and type-safe linkage, the C++ compiler normally appends `type` information to the names of functions. To prevent the C++ compiler from appending `type` information to the names of functions, and to allow FORTRAN to call functions, declare C++ functions with the `extern "C"` language construct. One common way to do this is in the declaration of a function:

```
extern "C" void abc ( int, float );
...
void abc ( int x, float y ) { /* ... body of abc ... */ }
```

For brevity, you can also combine `extern "C"` with the function definition, as in:

```
extern "C" void abc ( int x, float y )
{
        /* ... body of abc ... */
}
```

Most of the C++ examples in this chapter use this combined form. You cannot use the `extern "C"` language construct for member functions.

Array Indexing and Order

C++ arrays always start at zero, but by default, FORTRAN arrays start at 1. There are two common ways of approaching this.

- You can use the FORTRAN default, as in the above example. Then the FORTRAN element `B(2)` is equivalent to the C++ element `b[1]`.
- You can specify that the FORTRAN array `B` starts at `0`.

```
   INTEGER B(0:2)
```

  This way, the FORTRAN element `B(1)` is equivalent to the C element `b[1]`.

FORTRAN arrays are stored in column-major order, C++ arrays in row-major order. For one-dimensional arrays, this is no problem. For two-dimensional arrays, this is only a minor problem, as long as the array is square. Sometimes it is enough to just switch subscripts.

For two-dimensional arrays that are *not* square, it is not enough to just switch subscripts.

When linking libraries, note difference between Solaris 2.*x* and 1.*x*.

Library Linking On Solaris 2.x

(Solaris 2.x only) On Solaris 2.*x*, always use `CC` to link FORTRAN object files with C++ object files. You must link FORTRAN libraries explicitly:

```
% f77 -c -silent f1.f
% CC -c c1.cc
% CC f1.o c1.o -lM77 -lF77  ← Use CC to link.
```

Your main program can be either a FORTRAN program or a C++ program.

Linking Libraries On Solaris 1.x

(Solaris 1.x only) On Solaris 1.*x*, the main program must be in C++:

```
% f77 -c fsub.f
% CC -c main.cc
% CC main.o fsub.o -lM77 -lf77  ← Use CC to link.
```

If the main program must be in FORTRAN, you can use `f77` to link. In this case, however, your C++ code cannot include any static constructors, static destructors, exceptions, or templates. In addition, you must link the C++ library `libC` statically:

```
% f77 -c main.f
% CC -c cfun.cc
% f77 main.o cfun.o -Bstatic -lC -Bdynamic  ← Link libC statically.
```

File Descriptors and `stdio`

FORTRAN I/O channels are in terms of unit numbers. The I/O system does not deal with unit numbers, but with file descriptors. The FORTRAN runtime system translates from one to the other, so most FORTRAN programs don't have to know about file descriptors. Many C++ programs use a set of subroutines called standard I/O (or `stdio`). Many functions of FORTRAN I/O use standard I/O, which in turn uses operating system I/O calls.

Some of the characteristics of these I/O systems are:

Table 6-2 Characteristics of Three I/O Systems

	FORTRAN Units	**Standard I/O File Pointers**	**File Descriptors**
Files Open	Opened for reading and writing	Opened for reading, or opened for writing, or opened for both, or opened for appending see `OPEN`(3S)	Opened for reading, or opened for writing, or opened for both
Attributes	Formatted or unformatted	Always unformatted, but can be read or written with format-interpreting routines	Always unformatted
Access	Direct or sequential	Direct access if the physical file representation is direct access, but can always be read sequentially	Direct access if the physical file representation is direct access, but can always be read sequentially
Structure	Record	Character stream	Character stream
Form	Arbitrary, nonnegative integers	Pointers to structures in the user's address space	Integers from 0-63

File Permissions

C++ programmers traditionally open input files for reading and output files for writing, sometimes for both. In FORTRAN, the system cannot foresee what use you will make of the file since there's no parameter to the `OPEN` statement that gives that information.

FORTRAN tries to `OPEN` a file with the maximum permissions possible:

- First for both reading and writing
- Then for each separately.

This process takes place transparently and should be of concern only if you try to perform a `READ`, `WRITE`, or `ENDFILE`, but you do not have permission. Magnetic tape operations are an exception to this general freedom, since you could have write permissions on a file but not a write ring on the tape.

6

FORTRAN Calls C++

This section describes the interface when FORTRAN calls C++.

Arguments Passed by Reference

There are two types of arguments: simple types and complex types.

Simple Types

For simple types, define each C++ argument as a reference.

Code Example 6-2 Passing Arguments by Reference—C++ Program

SimRef.cc	

```
extern "C" void simref (
    char& t,
    char& f,
    char& c,
    int& i,
    float& r,
    double& d,
    short& si )
{
    t = 1;
    f = 0;
    c = 'z';
    i = 9;
    r = 9.9;
    d = 9.9;
    si = 9;
}
```

Default: Pass each FORTRAN argument by reference.

Code Example 6-3 Passing Arguments by Reference—FORTRAN Program

SimRefmain.f

```
      logical*1 t, f
      character c
      integer*4 i
      real r
      double precision d
      integer*2 si
      external SimRef !$pragma C( SimRef )
      call SimRef ( t, f, c, i, r, d, si )
      write( *, "(L2,L2,A2,I2,F4.1,F4.1,I2)" )
&   t,f,c,i,r,d,si
      end
```

Compile and execute, with output, as follows:

```
% CC -c SimRef.cc
% f77 -c -silent SimRefmain.f
% CC SimRef.o SimRefmain.o -lM77 -lF77
% a.out
 T F z 9 9.9 9.9 9
```

Complex Types

Here, the C++ argument is a pointer to a structure.

Code Example 6-4 Passing Arguments by Reference—FORTRAN Calls C++

CmplxRef.cc	

```
struct complex { float r, i; };
struct dcomplex { double r, i; };

extern "C" void cmplxref ( complex& w, dcomplex& z ) {
    w.r = 6;
    w.i = 7;
    z.r = 8;
    z.i = 9;
}
```

CmplxRefmain.f	

```
      complex w
      double complex z
      external CmplxRef !$pragma C ( CmplxRef )
      call CmplxRef( w, z )
      write(*,*) w
      write(*,*) z
      end
```

Compile and execute, with output.

```
% CC -c CmplxRef.cc
% f77 -c -silent CmplxRefmain.f
% CC CmplxRef.o CmplxRefmain.o -lM77 -lF77
% a.out
  ( 6.00000, 7.00000)
  ( 8.0000000000000, 9.0000000000000)
```

A C++ reference to a `float` matches a `REAL` passed by reference.

Character Strings Passed by Reference

Passing strings between C++ and FORTRAN is not recommended.

For every FORTRAN argument of character type, FORTRAN associates an extra argument, giving the length of the string. The string lengths are equivalent to C++ `long int` quantities passed by value. In standard C++ use, all C++ strings are passed by reference. The order of arguments is:

- Address for each argument (datum or function)
- The length of each character argument, as a `long int`.

The whole list of string lengths comes after the whole list of other arguments.

The FORTRAN call in:

```
    CHARACTER*7 S
    INTEGER B(3)
(arguments)
    CALL SAM( B(2), S )
```

is equivalent to the C++ call in:

```
    char s[7];
    long int b[3];
(arguments)
    sam_( &b[1], s, 7L );
```

Ignoring the Extra Arguments

You can ignore these extra arguments, since they are after the list of other arguments.

Code Example 6-5 Passing Strings by Reference—Ignoring the Extra Arguments

StrRef.cc	#include <string.h>
	
	extern "C" void strref (char (&s10)[], char (&s26)[]) {
	  static char ax[11] = "abcdefghij";
	  static char sx[27] = "abcdefghijklmnopqrstuvwxyz";
	
	  strncpy(s10, ax, 10);
	  strncpy(s26, sx, 26);
	}

StrRefmain.f	character s10*10, s26*26 external StrRef !$pragma C(StrRef) Call StrRef(s10, s26) write(*, 1) s10, s26 1 format("s10='", A, "'", / "s26='", A, "'") end

Compile and execute, with output:

```
% CC -c StrRef.cc
% f77 -c -silent StrRefmain.f
% CC StrRef.o StrRefmain.o -lM77 -lF77
% a.out
s10='abcdefghij'
s26='abcdefghijklmnopqrstuvwxyz'
```

Using the Extra Arguments

You can use the extra arguments. In the following example, all this C++ function does with the lengths is print them:

Code Example 6-6 Passing Strings by Reference—Using the Extra Arguments

StrRef2.cc

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

extern "C" void strref ( char (&s10)[], char (&s26)[], int L10,
int L26 ) {
    static char ax[11] = "abcdefghij";
    static char sx[27] = "abcdefghijklmnopqrstuvwxyz";
    printf( "%d %d\n", L10, L26 );
    strncpy( s10, ax, 10 );
    strncpy( s26, sx, 26 );
}
```

Compile and execute, with output:

```
% CC -c StrRef2.cc
% f77 -c -silent StrRefmain.f
% CC StrRef2.o StrRefmain.o -lM77 -lF77
% a.out
10 26
s10='abcdefghij'
s26='abcdefghijklmnopqrstuvwxyz'
```

One-Dimensional Arrays Passed by Reference

Code Example 6-7 shows a C++ array, indexed from 0 to 8:

Code Example 6-7 Passing Arrays by Reference (C++ code)

FixVec.cc

```
extern "C" void fixvec ( int V[9], int& Sum )
{
    Sum= 0;
    for( int i= 0; i < 9; ++i ) {
        Sum += V[i];
    }
}
```

Code Example 6-8 shows a FORTRAN array, implicitly indexed from 1 to 9:

Code Example 6-8 Passing Arrays by Reference (FORTRAN code)

FixVecmain.f

```
      integer i, Sum
      integer a(9) / 1,2,3,4,5,6,7,8,9 /
      external FixVec !$pragma C( FixVec )
      call FixVec( a, Sum )
      write( *, '(9I2, " ->" I3)') (a(i),i=1,9), Sum
      end
```

Compile and execute, with output:

```
% CC -c FixVec.cc
% f77 -c -silent FixVecmain.f
% CC FixVec.o FixVecmain.o -lM77 -lF77
% a.out
 1 2 3 4 5 6 7 8 9 -> 45
```

Two-Dimensional Arrays Passed by Reference

In a two-dimensional array, the rows and columns are switched. Such square arrays are either incompatible between C++ and FORTRAN, or awkward to keep straight. Non-square arrays are even more difficult to maintain.

Code Example 6-9 shows a 2 by 2 C++ array, indexed from 0 to 1, and 0 to 1.

Code Example 6-9 A Two-dimensional C++ Array

FixMat.cc	extern "C" void fixmat (int a[2][2])
	{
	    a[0][1] = 99;
	}

Code Example 6-10 shows a 2 by 2 FORTRAN array, explicitly indexed from 0 to 1, and 0 to 1.

Code Example 6-10 A Two-dimensional FORTRAN Array

FixMatmain.f	integer c, m(0:1,0:1) / 00, 10, 01, 11 /, r
	external FixMat !$pragma C(FixMat)
	do r= 0, 1
	   do c= 0, 1
	      write(*, '("m(",I1,",",I1,")=",I2.2)') r, c, m(r,c)
	   end do
	end do
	
	call FixMat(m)
	write(*, *)
	
	do r= 0, 1
	   do c= 0, 1
	      write(*, '("m(",I1,",",I1,")=",I2.2)') r, c, m(r,c)
	   end do
	end do
	
	end

Compile and execute. Show `m` before and after the C call.

```
% CC -c FixMat.cc
% f77 -c -silent FixMatmain.f
% CC FixMat.o FixMatmain.o -lM77 -lF77
% a.out
m(0,0) = 00
m(0,1) = 01
m(1,0) = 10
m(1,1) = 11

m(0,0) = 00
m(0,1) = 01
m(1,0) = 99
m(1,1) = 11
```

Compare `a[0][1]` with `m(1,0)`: C++ changes `a[0][1]`, which is FORTRAN `m(1,0)`.

Structured Records Passed by Reference

Code Example 6-11 and Code Example 6-12 show how to pass a structure to FORTRAN:

Code Example 6-11 Passing Structures to FORTRAN (C++ Code)

StruRef.cc

```
struct VarLenStr {
    int nbytes ;
    char a[26];
};

#include <stdlib.h>
#include <string.h>
extern "C" void struchr ( VarLenStr& v )
{
    memcpy(v.a, "oyvay", 5);
    v.nbytes= 5;
}
```

Code Example 6-12 Passing Structures to FORTRAN (FORTRAN Code)

StruRefmain.f

```
      structure /VarLenStr/
          integer nbytes
          character a*25
      end structure
      record /VarLenStr/ vls
      character s25*25

      external StruChr !$pragma C(StruChr)

      vls.nbytes= 0
      call StruChr( vls )
      s25(1:5) = vls.a(1:vls.nbytes)
      write(*, 1 ) vls.nbytes, s25
 1    format( "size =", I2, ", s25='", A, "'" )
      end
```

Compile and execute, with output:

```
% CC -c StruRef.cc
% f77 -c -silent StruRefmain.f
% CC StruRef.o StruRefmain.o -lM77 -lF77
% a.out
size = 5, s25='oyvay'
```

Pointers Passed by Reference

C++ gets a reference to a pointer, as follows:

Code Example 6-13 Passing Pointers by Reference (C++ Code)

PassPtr.cc

```
extern "C" void passptr ( int* & i, double* & d )
{
    *i = 9;
    *d = 9.9;
}
```

Code Example 6-14 shows how FORTRAN passes the pointer by reference:

Code Example 6-14 Passing Pointers by Reference (FORTRAN code)

PassPtrmain.f

```
      program PassPtrmain
      integer i
      double precision d
      pointer (iPtr, i), (dPtr, d)
      external PassPtr !$pragma C ( PassPtr )
      iPtr = malloc( 4 )
      dPtr = malloc( 8 )
      i = 0
      d = 0.0
      call PassPtr( iPtr, dPtr )
      write( *, "(i2, f4.1)" ) i, d
      end
```

Compile and execute, with output:

```
% CC -c PassPtr.cc
% f77 -c -silent PassPtrmain.f
% CC PassPtr.o PassPtrmain.o -lM77 -lF77
% a.out
 9 9.9
```

Arguments Passed by Value

In the call, enclose an argument in the nonstandard function %VAL().

Simple Types Passed by Value

Code Example 6-15 and Code Example 6-16 show how to pass simple types by value.

Code Example 6-15 Passing Simple Types by Value (C++ Code)

SimVal.cc

```
extern "C" void simval (
    char   t,
    char   f,
    char   c,
    int    i,
    double d,
    short  s,
    int&   reply )
{
    reply= 0;
    // If nth arg ok, set nth octal digit to one
    if( t        ) reply +=      01;
    if( ! f      ) reply +=     010;
    if( c == 'z' ) reply +=    0100;
    if( i == 9   ) reply +=   01000;
    if( d == 9.9 ) reply +=  010000;
    if( s == 9   ) reply += 0100000;
}
```

Code Example 6-16 Passing Simple Types by Value (FORTRAN Code)

SimValmain.f

```
      logical*1 t, f
      character c
      integer*4 i
      double precision d
      integer*2 s
      integer*4 args

      data t/.true./, f/.false./, c/'z'/
&   i/9/, d/9.9/, s/9/

      external SimVal !$pragma C( SimVal )
      call SimVal ( %VAL(t), %VAL(f), %VAL(c),
&   %VAL(i), %VAL(d), %VAL(s), args )
      write( *, 1 ) args
 1    format( 'args=', o6, ' (If nth digit=1, arg n OK)' )
      end
```

Pass each FORTRAN argument by value, except for `args`. The same rule applies to `CHARACTER*1`, `COMPLEX`, `DOUBLE COMPLEX`, `INTEGER`, `LOGICAL`, `DOUBLE PRECISION`, structures, and pointers.

Compile and execute, with output:

```
% CC -c SimVal.cc
% f77 -c -silent SimValmain.f
% CC SimVal.o SimValmain.o -lM77 -lF77
% a.out
args=111111(If nth digit=1, arg n OK)
```

Real Variables Passed by Value

Real Variables are passed by value the same way other simple types are. Code Example 6-17 shows how to pass a real variable:

Code Example 6-17 Passing a Real Variable

FloatVal.cc

```
#include <math.h>

extern "C" void floatval ( float f, double& d ) {
    float x=f;
    d = double(x) + 1.0 ;
}
```

FloatValmain.f

```
      double precision d
      real r / 8.0 /
      external FloatVal !$pragma C( FloatVal )
      call FloatVal( %VAL(r), d )
      write( *, * ) r, d
      end
```

Compile and execute, with output:

```
% CC -c FloatVal.cc
% f77 -c -silent FloatValmain.f
% CC FloatVal.o FloatValmain.o -lM77 -lF77
% a.out
    8.00000 9.0000000000000
```

Complex Types Passed by Value

You can pass the `complex` structure by value, as Code Example 6-18 shows:

Code Example 6-18 Passing Complex Types

CmplxVal.cc	

	`struct complex { float r, i; };`
	
	`extern "C" void cmplxval ( complex  w, complex& z ) {`
	`  z.r = w.r * 2.0 ;`
	`  z.i = w.i * 2.0 ;`
	`  w.r = 0.0 ;`
	`  w.i = 0.0 ;`
	`}`

CmplxValmain.f	

	`  complex w / (4.0, 4.5 ) /`
	`  complex z`
	`  external CmplxVal !$pragma C( CmplxVal )`
	`  call CmplxVal( %VAL(w), z )`
	`  write ( *, * ) w`
	`  write ( *, * ) z`
	`  end`

Compile and execute, with output

```
% CC -c CmplxVal.cc
% f77 -c -silent CmplxValmain.f
% CC CmplVal.o CmplxValmain.o -lM77 -lF77
% a.out
  ( 4.00000, 4.50000)
  ( 8.00000, 9.00000)
```

Arrays, Strings, Structures Passed by Value - N/A

There is no reliable way to pass arrays, character strings, or structures by value on all architectures. The workaround is to pass them by reference.

Pointers Passed by Value

C++ gets a pointer.

Code Example 6-19 Passing Pointers by Value (C++ Code)

PassPtrVal.cc

```
extern “C” void passptrval ( int* i, double* d )
{
    *i = 9;
    *d = 9.9;
}
```

FORTRAN passes a pointer by value:

Code Example 6-20 Passing Pointers by Value (FORTRAN Code)

PassPtrValmain.f

```
      program PassPtrValmain
      integer i
      double precision d
      pointer (iPtr, i), (dPtr, d)
      external PassPtrVal !$pragma C ( PassPtrVal )
      iPtr = malloc( 4 )
      dPtr = malloc( 8 )
      i = 0
      d = 0.0
      call PassPtrVal( %VAL(iPtr), %VAL(dPtr) ) ! Nonstandard?
      write( *, "(i2, f4.1)" ) i, d
      end
```

Compile and execute, with output:

```
% CC -c PassPtrVal.cc
% f77 -c -silent PassPtrValmain.f
% CC PassPtrVal.o PassPtrValmain.o -lM77 -lF77
% a.out
 9 9.9
```

Function Return Values

For function return values, a FORTRAN function of type `BYTE`, `INTEGER`, `REAL`, `LOGICAL`, or `DOUBLE PRECISION` is equivalent to a C++ function that returns the corresponding type. There are two extra arguments for the return values of character functions, and one extra argument for the return values of complex functions.

int

Code Example 6-21 shows how to return an `int` to a FORTRAN program.

Code Example 6-21 Returning an `int` to FORTRAN

RetInt.cc

```
extern "C" int retint ( int& r )
{
    int s;
    s = r;
    ++s;
    return s;
}
```

RetIntmain.f

```
      integer r, s, RetInt
      external RetInt !$pragma C( RetInt )
      r = 2
      s = RetInt( r )
      write( *, "(2I4)") r, s
      end
```

Compile, link, and execute, with output.

```
% CC -c RetInt.cc
% f77 -c -silent RetInt.o RetIntmain.f
% CC RetInt.o RetIntmain.o -lM77 -lF77
% a.out
 2 3
%
```

Do a function of type `BYTE`, `LOGICAL`, `REAL`, or `DOUBLE PRECISION` in the same way. Use matching types according to Table 6-1 on page 108.

float

Code Example 6-22 shows how to return a `float` to a FORTRAN program:

Code Example 6-22 Return a `float` to FORTRAN

RetFloat.cc	

	`extern "C" float retfloat ( float& pf )`
	`{`
	`    float f;`
	`    f = pf;`
	`    ++f;`
	`    return f;`
	`}`

RetFloatmain.f	`      real RetFloat, r, s`
	`      external RetFloat !$pragma C( RetFloat )`
	`      r = 8.0`
	`      s = RetFloat( r )`
	`      print *, r, s`
	`      end`

```
% CC -c RetFloat.cc
% f77 -c -silent RetFloatmain.f
% CC RetFloat.o RetFloatmain.o -lM77 -lF77
% a.out
    8.00000 9.00000
```

A Pointer to a `float`

Code Example 6-23 shows how to return a function value that is a pointer to a `float`.

Code Example 6-23 Return a pointer to a `float` to FORTRAN

RetPtrF.cc	

	`static float f;`
	
	`extern "C" float* retptrf ( float& a )`
	`{`
	`    f = a;`
	`    ++f;`
	`    return &f;`
	`}`

Code Example 6-23 Return a pointer to a `float` to FORTRAN

RetPtrFmain.f

```
      integer RetPtrF
      external RetPtrF !$pragma C( RetPtrF )
      pointer (p, s)
      real r, s
      r = 8.0
      p = RetPtrF( r )
      print *, s
      end
```

Compile and execute, with output:

```
% CC -c RetPtrF.cc
% f77 -c -silent RetPtrFmain.f
% CC RetPtrF.o RetPtrFmain.o -lM77 -lF77
% a.out
9.00000
```

The function return value is an address; you can assign it to the pointer value, or do some pointer arithmetic. You cannot use it in an expression with `real`s, such as `RetPtrF(R)+100.0`.

Double Precision

Code Example 6-24 is an example of C++ returning a type `double` function value to a FORTRAN `DOUBLE PRECISION` variable:

Code Example 6-24 Return a `double` to FORTRAN

RetDbl.cc

```
extern "C" double retdbl ( double& r )
{
    double s;
    s = r;
    ++s;
    return s;
}
```

Code Example 6-24 Return a `double` to FORTRAN

RetDblmain.f

```
      double precision r, s, RetDbl
      external RetDbl !$pragma C( RetDbl )
      r = 8.0
      s = RetDbl( r )
      write( *, "(2F6.1)" ) r, s
      end
```

Compile and execute, with output.

```
% CC -c RetDbl.cc
% f77 -c -silent RetDblmain.f
% CC RetDbl.o RetDblmain.o -lM77 -lF77
% a.out
   8.0 9.0
```

complex

A `COMPLEX` or `DOUBLE COMPLEX` function is equivalent to a C++ routine having an additional initial argument that points to the return value storage location. A general pattern for such a FORTRAN function is:

```
    COMPLEX FUNCTION F ( arguments )
```

The pattern for a corresponding C++ function is:

```
struct complex { float r, i; };
f_ ( complex temp, arguments );
```

Code Example 6-25 shows how to return a type `COMPLEX` function value to FORTRAN.

Code Example 6-25 Returning a COMPLEX value to FORTRAN

RetCmplx.cc

```
struct complex { float r, i; };

extern "C" void retcmplx ( complex& RetVal, complex& w ) {
    RetVal.r = w.r + 1.0 ;
    RetVal.i = w.i + 1.0 ;
    return;
}
```

Code Example 6-25 Returning a COMPLEX value to FORTRAN

RetCmplxmain.f

```
      complex u, v, RetCmplx
      external RetCmplx !$pragma C( RetCmplx )
      u = ( 7.0, 8.0 )
      v = RetCmplx( u )
      write( *, * ) u
      write( *, * ) v
      end
```

Compile and execute, with output:

```
% CC -c -silent RetCmplx.cc
% f77 -c -silent RetCmplxmain.f
% CC RetCmplx.o RetCmplxmain.o -lM77 -lF77
% a.out
  ( 7.00000, 8.00000)
  ( 8.00000, 9.00000)
```

Character Strings

Passing strings between C++ and FORTRAN is not recommended. A character-string-valued FORTRAN function is equivalent to a C++ function with the two extra initial arguments—data address and length. A FORTRAN function of this form:

```
    CHARACTER*15 FUNCTION G (arguments)
```

and a C++ function of this form:

```
g_ (char * result, ling int length, other arguments)
char result[ ];
long int length;
```

are equivalent and can be invoked in C++ with this call:

```
char chars[15];
(arguments)
    g_ (chars, 15L, other arguments);
```

Code Example 6-26 shows how to return a character string to a FORTRAN program.

Code Example 6-26 Returning a String to FORTRAN (C++ Code)

RetStr.cc

```
#include <stdio.h>

extern "C" void retstr_ ( char *retval_ptr, int retval_len,
    char& ch_ref,
    int& n_ref,
    int  ch_len )
{
    int count = n_ref;
    char *cp = retval_ptr;
    for( int i= 0; i < count; ++i ) {
    *cp++ = ch_ref;
    }
}
```

- The returned string is passed by the extra arguments `retval_ptr` and `retval_len`, a pointer to the start of the string and the string's length.
- The character-string argument is passed with `ch_ptr` and `ch_len`.
- The `ch_len` is at the end of the argument list.
- The repeat factor is passed as `n_ptr`.

In FORTRAN, use the above C++ function as shown here:

Code Example 6-27 Returning a String to FORTRAN (FORTRAN Code)

RetStrmain.f

```
      character String*100, RetStr*50
      String = RetStr( '*', 10 )
      print *, "'", String(1:10), "'"
      end
```

Compile and execute with output:

```
% CC -c RetStr.cc
% f77 -c -silent RetStrmain.f
% CC RetStr.o RetStrmain.o -lM77 -lF77
% a.out
'**********'
```

6

Labeled Common

C++ and FORTRAN can share values in labeled common. The method is the same no matter which language calls which, as shown by Code Example 6-28 and Code Example 6-29:

Code Example 6-28 Using Labeled Common (FORTRAN Code)

UseCom.f

```
    subroutine UseCom ( n )
    integer n
    real u, v, w
    common / ilk / u, v, w
    n = 3
    u = 7.0
    v = 8.0
    w = 9.0
    return
    end
```

Code Example 6-29 Using Labeled Common (C++ Code)

UseCommain.cc

```
#include <stdio.h>
extern struct comtype {
    float p;
    float q;
    float r;
};
extern struct comtype ilk_;
main()
{
    char *string = "abc0";
    int count = 3;
    extern void usecom_ ( );
    ilk_.p = 1.0;
    ilk_.q = 2.0;
    ilk_.r = 3.0;
    usecom_ ( string, count );
    printf(" ilk_.p=%4.1f, ilk_.q=%4.1f, ilk_.r=%4.1f\n",
    ilk_.p, ilk_.q, ilk_.r );
}
```

Compile and execute, with output:

```
% f77 -c -silent UseCom.f
% CC -c UseCommain.cc
% CC UseCom.o UseCommain.o -lF77 -lM77
% a.out
 ilk_p = 7.0, ilk_q = 8.0, ilk_r = 9.0
```

Any of the options that change size, alignment, or any equivalences that change alignment may invalidate such sharing.

I/O Sharing

It's not a good idea to mix FORTRAN I/O with C++ I/O. If you must mix them, it's safer to pick one and not alternate.

The FORTRAN I/O library is implemented largely on top of the C standard I/O library. Every open unit in a FORTRAN program has an associated standard I/O file structure. For the `stdin`, `stdout`, and `stderr` streams, the file structure need not be explicitly referenced, so it is possible to share them. However, the C++ stream I/O system uses a different mechanism.

If a FORTRAN main program calls C++, then before the FORTRAN program starts, the FORTRAN I/O library is initialized to connect units 0, 5, and 6 to `stderr`, `stdin`, and `stdout`, respectively. The C++ function must take the FORTRAN I/O environment into consideration to perform I/O on open file descriptors.

`stdout`

Code Example 6-30 shows a C++ function that writes to `stderr` and to `stdout`, and the FORTRAN code that calls the C++ function:

Code Example 6-30 Mixing with `stdout`

MixIO.cc

```
#include <stdio.h>

extern "C" void mixio ( int& n ) {
    if( n <= 0 ) {
    fprintf( stderr, "Error:  negative line number (%d)\n", n );
    n= 1;
    }
    printf( "In   C++:    line # = %2d\n", n );
}
```

MixIOmain.f

```
      integer n/ -9 /
      external MixIO !$pragma C( MixIO )
      do i= 1, 6
         n = n +1
         if ( abs(mod(n,2)) .eq. 1 ) then
            call MixIO( n )
         else
            write( *, '("In Fortran:  line # = ", i2)' ) n
         end if
      end do
      end
```

Compile and execute, with output:

```
% CC -c MixIO.cc
% f77 -c -silent MixIOmain.f
% CC MixIO.o MixIOmain.o -lM77 -lF77
% a.out
In FORTRAN: line # =-8
error: negative line #
In C: line # = 1
In FORTRAN: line # = 2
In C: line # = 3
In FORTRAN: line # = 4
In C: line # = 5
```

stdin

Code Example 6-31 shows a C++ function that reads from `stdin`, and the FORTRAN code that calls the C++ function:

Code Example 6-31 Mixing with `stdin`

MixStdin.cc

```
#include <stdio.h>

extern "C" int c_read_ ( FILE* &fp, char *buf, int& nbytes, int
buf_len )
{
    return fread( buf, 1, nbytes, fp );
}
```

MixStdinmain.f

```
      character*1 inbyte
      integer*4 c_read, getfilep
      external getfilep
      write( *, '(a, $)') 'What is the digit? '
      flush (6)
      irtn = c_read( getfilep( 5 ), inbyte, 1 )
      write( *, 9 ) inbyte
 9    format( 'The digit read by C++ is ', a )
      end
```

FORTRAN does the prompt; C++ does the read, as follows:

```
demo% CC -c MixStdin.cc
demo% f77 -c -silent MixStdinmain.f
demo% CC MixSdin.o MixStdinmain.o -lM77 -lF77
demo% a.out
What is the digit? 3
The digit read by C is 3
demo%■
```

Alternate Returns - N/A

C++ does not have an alternate return. The workaround is to pass an argument and branch on that.

6

C++ Calls FORTRAN

This section describes the interface when C++ calls FORTRAN.

Arguments Passed by Reference

Simple Variables Passed by Reference

Here, FORTRAN expects all these arguments to be passed by reference, which is the default.

Code Example 6-32 Passing Variables by Reference (FORTRAN Code)

SimRef.f	
	subroutine SimRef (t, f, c, i, d, si, sr) logical*1 t, f character c integer i double precision d integer*2 si real sr t = .true. f = .false. c = 'z' i = 9 d = 9.9 si = 9 sr = 9.9 return end

Here, C++ passes the address of each.

Code Example 6-33 Passing Variables by Reference (C++ Code)

SimRefmain.cc

```
extern "C" void simref_( char&, char&, char&, int&, double&,
short&, float& );
#include <stdio.h>
#include <stdlib.h>

main ( ) {
    char t, f, c;
    int i;
    double d;
    short si;
    float sr;

    simref_( t, f, c, i, d, si, sr );
    printf( "%08o %08o %c %d %3.1f %d %3.1f\n",
       t, f, c, i, d, si, sr );
    return 0;
}
```

Almost all examples in this manual of C++ calling FORTRAN use the command for dynamic linking for simplicity. We suggest you use dynamic linking.

Compile, link dynamically, and execute, with output:

```
demo% f77 -c -silent SimRef.f
demo% CC -c SimRefmain.cc
demo% CC SimRef.o SimRefmain.o -lM77 -lF77  ← This does the linking.
demo% a.out
00000001 00000000 z 9 9.9 9 9.9
demo% ■
```

In this example, the second `f77` command does dynamic linking.

Complex Variables Passed by Reference

The complex types require a simple structure as shown in Code Example 6-34. In this example `w` and `z` are passed by reference, which is the default:

Code Example 6-34 Passing Complex Variables

CmplxRef.f	

	`      subroutine CmplxRef ( w, z )`
	`      complex w`
	`      double complex z`
	`      w = ( 6, 7 )`
	`      z = ( 8, 9 )`
	`      return`
	`      end`

CmplxRefmain.cc	`#include <stdlib.h>`
	`#include <stdio.h>`
	
	`struct complex { float r, i; };`
	`struct dcomplex { double r, i; };`
	
	`extern "C" void cmplxref_ ( complex& w, dcomplex& z );`
	
	`main ( ) {`
	`    complex d1;`
	`    dcomplex d2;`
	
	`    cmplxref_( d1, d2 );`
	`   printf( "%3.1f %3.1f\n%3.1f %3.1f\n", d1.r, d1.i, d2.r, d2.i );`
	`    return 0;`
	`}`

The following example shows `CmplxRef.f` compiled and executed with output:

```
% f77 -c -silent CmplxRef.f
% CC -c CmplxRefmain.cc
% CC CmplxRef.o CmplxRefmain.o -lF77 -lM77
% a.out
6.0 7.0
8.0 9.0
```

Character Strings Passed by Reference

Character strings match in a straightforward manner. If you make the string in FORTRAN, you must provide the explicit null terminator because FORTRAN does not automatically do that, and C++ expects it.

Note – We suggest you avoid passing strings between C++ and FORTRAN.

Code Example 6-35 Passing Strings by Reference

StrRef.f

```
      subroutine StrRef ( a, s )
      character a*10, s*80
      a = 'abcdefghi' // char(0)
      s = 'abcdefghijklmnopqrstuvwxyz' // char(0)
      return
      end
```

StrRefmain.cc

```
#include <stdlib.h>
#include <stdio.h>

extern "C" void strref_ ( char*, char* );

main ( ) {
    char s10[10], s80[80];

    strref_( s10, s80 );
    printf( " s10='%s'\n s80='%s'\n", s10, s80 );
    return 0;
}
```

Compile and execute, with output:

```
% f77 -c -silent StrRef.f
% CC -c StrRefmain.cc
% CC StrRef.o StrRefmain.o -lM77 -lF77
% a.out
s10='abcdefghi'
s80='abcdefghijklmnopqrstuvwxyz'
```

Arguments Passed by Value - N/A

FORTRAN can call C++ and pass an argument by value. However FORTRAN cannot handle an argument passed by value. The workaround is to pass all arguments by reference.

Function Return Values

For function return values, a FORTRAN function of type `BYTE`, `INTEGER`, `LOGICAL`, or `DOUBLE PRECISION` is equivalent to a C++ function that returns the corresponding type. There are two extra arguments for the return values of character functions and one extra argument for the return values of complex functions.

`int`

Code Example 6-36 shows how FORTRAN returns an `INTEGER` function value to C++:

Code Example 6-36 Return an Integer to C++

`RetInt.f`

```
      integer function RetInt ( k )
      integer k
      RetInt = k + 1
      return
      end
```

`RetIntmain.cc`

```
#include <stdio.h>
#include <stdlib.h>

extern "C" int retint_ ( int& );

main ( ) {
      int k = 8;
      int m = retint_( k );
      printf( "%d %d\n", k, m );
      return 0;
}
```

Compile and execute, with output:

```
% f77 -c -silent RetInt.f
% CC -c RetIntmain.cc
% CC RetInt.o RetIntmain.o -lM77 -lF77
% a.out
8 9
```

float

Code Example 6-37 shows how to return a `float` to a C++ program.

Code Example 6-37 Returning a `float` to C++

RetFloat.f

```
      real function RetReal ( x )
      real x
      RetReal = x + 1.0
      return
      end
```

RetFloatmain.cc

```
#include <stdio.h>
extern "C" float retreal_ (float*) ;
main ( )
{
        float r, s ;
        r = 8.0 ;
        s = retreal_ ( &r ) ;
        printf( " %8.6f %8.6f \n", r, s ) ;
        return 0;
}
```

Compile and execute, with output:

```
% f77 -c -silent RetFloat.f
% CC -c -w RetFloatmain.cc
% CC RetFloat.o RetFloatmain.o -lM77 -lF77
% a.out
 8.000000 9.000000
```

double

Code Example 6-38 shows how FORTRAN returns a `DOUBLE PRECISION` function value to C++.

Code Example 6-38 Returning a `double` to C++

RetDbl.f	

	`      double precision function RetDbl ( x )`
	`      double precision x`
	`      RetDbl = x + 1.0`
	`      return`
	`      end`

RetDblmain.cc	

	`#include <stdio.h>`
	`#include <stdlib.h>`
	
	`extern "C" double retdbl_ ( double& );`
	
	`main ( ) {`
	`      double x = 8.0;`
	`      double y = retdbl_( x );`
	`      printf( "%8.6f %8.6f\n", x, y );`
	`      return 0;`
	`}`

Compile and execute, with output.

```
% f77 -c -silent RetDbl.f
% CC -c RetDblmain.cc
% CC RetDbl.o RetDblmain.o -lM77 -lF77
% a.out
8.000000 9.000000
```

COMPLEX *or* DOUBLE COMPLEX

A `COMPLEX` or `DOUBLE COMPLEX` function is equivalent to a C++ routine having an additional initial argument that points to the return value storage location. A general pattern for such a FORTRAN function is:

```
    COMPLEX FUNCTION F ( arguments )
```

The pattern for a corresponding C++ function is

```
struct complex { float r, i; };
void f_ ( complex &, other arguments )
```

Code Example 6-39 shows how to return a COMPLEX:

Code Example 6-39 Returning a COMPLEX

RetCmplx.f

```
      complex function RetCmplx ( x )
      complex x
      RetCmplx = x * 2.0
      return
      end
```

RetCmplxmain.cc

```
#include <stdlib.h>
#include <stdio.h>
struct complex { float r, i; };

extern "C" void retcmplx_( complex&, complex& );

main ( ) {
    complex c1, c2 ;

    c1.r = 4.0;
    c1.i = 4.5;

    retcmplx_( c2, c1 );

   printf( " %3.1f %3.1f\n %3.1f %3.1f\n", c1.r, c1.i, c2.r, c2.i
);
    return 0;
}
```

Compile, link, and execute, with output:

```
% f77 -c -silent RetCmplx.f
% CC -c -w RetCmplxmain.cc
% CC RetCmplx.o RetCmplxmain.o -lM77 -lF77
% a.out
 4.0 4.5
 8.0 9.0
```

When you use f77 to pass files to the linker, the linker uses the f77 libraries.

Character Strings

Note – We suggest you avoid passing strings between C++ and FORTRAN.

A FORTRAN string function has two extra initial arguments: data address and length. If you have a FORTRAN function of the following form:

```
    CHARACTER*15 FUNCTION G ( arguments )
```

and a C++ function of this form:

```
g_ ( char * result, long int length, other arguments )
```

they are equivalent, and can be invoked in C++ with:

```
char chars[15];
g_ ( chars, 15L, arguments );
```

The lengths are passed by value. You must provide the null terminator. Code Example 6-40 shows how to pass a string to C++.

Code Example 6-40 Returning a String to C++

RetChr.f

```
      function RetChr( c, n )
      character RetChr*(*), c
      RetChr = ''
      do i = 1, n
         RetChr(i:i) = c
      end do

      RetChr(n+1:n+1) = char(0) ! Put in the null terminator.
      return
      end
```

Code Example 6-40 Returning a String to C++

RetChrmain.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern "C" void retchr_( char*, int, char*, int&, int );

main ( ) {
    char string[100], repeat_val[50];

    int repeat_len = sizeof( repeat_val );
    int count = 10;

    retchr_( repeat_val, repeat_len, "*", count, sizeof("*")-1 );

    strncpy( string, repeat_val, repeat_len );
    printf( " '%s'\n", repeat_val );
    return 0;
}
```

Compile, link, and execute, with output.

```
% f77 -c -silent RetChr.f
% CC -c RetChrmain.cc
% CC RetChr.o RetChrmain.o -lM77 -lF77
% a.out
 '**********'
```

The caller must set up more arguments than are apparent as formal parameters to the FORTRAN function. Arguments that are lengths of character strings are passed by value. Those that are not are passed by reference.

Labeled Common

C++ and FORTRAN can share values in labeled common. Any of the options that change size, alignment, or any equivalences that change alignment may invalidate such sharing.

The method is the same, no matter which language calls which.

Code Example 6-41 Using Labeled Common (FORTRAN Code)

UseCom.f

```
      subroutine UseCom ( n )
      integer n
      real u, v, w
      common /ilk/ u, v, w
      n = 3
      u = 7.0
      v = 8.0
      w = 9.0
      return
      end
```

Code Example 6-42 Using Labeled Common (C++ Code)

UseCommain.cc

```
#include <stdio.h>

struct ilk_type {
    float p;
    float q;
    float r;
};

extern ilk_type ilk_ ;
extern "C" void usecom_ ( int& );

main ( ) {
    char *string = "abc0" ;
    int count = 3;
    ilk_.p = 1.0;
    ilk_.q = 2.0;
    ilk_.r = 3.0;
    usecom_( count );
    printf( " ilk_.p=%4.1f,  ilk_.q=%4.1f,  ilk_.r=%4.1f\n",
       ilk_.p, ilk_.q, ilk_.r );
    return 0;
}
```

Compile and execute, with output.

```
% f77 -c -silent UseCom.f
% CC -c UseCommain.cc
% CC UseCom.o UseCommain.o -lF77 -lM77
% a.out
 ilk_.p = 7.0, ilk_.q = 8.0, ilk_.r = 9.0
```

I/O Sharing

We do not recommend mixing FORTRAN I/O with C++ I/O. If you must mix them, it is usually safer to pick one and not alternate.

The FORTRAN I/O library uses the C++ standard I/O library. Every open unit in a FORTRAN program has an associated standard I/O file structure. For the `stdin`, `stdout`, and `stderr` streams, the file structure need not be explicitly referenced, so it is possible to share them.

For sharing I/O, if a C++ main program calls a FORTRAN subprogram, then there is no automatic initialization of the FORTRAN I/O library (connect units 0, 5, and 6 to `stderr`, `stdin`, and `stdout`, respectively). If a FORTRAN function attempts to reference the `stderr` stream (unit 0), then any output is written to a file named `fort.0` instead of to the `stderr` stream.

To make the C++ program initialize I/O and establish the preconnection of units 0, 5, and 6, insert the following line at the start of the C++ `main`.

```
    f_init();
```

At the end of the C++ `main`, you can insert:

```
    f_exit();
```

although it may not be necessary.

Code Example 6-43 and Code Example 6-44 show how to share I/O using a C++ main program and a FORTRAN subroutine.

Code Example 6-43 Sharing I/O (FORTRAN Code)

MixIO.f

```
      subroutine MixIO ( n )
      integer n
      if ( n .le. 0 ) then
         write(0,*) "error: negative line #"
         n = 1
      end if

      write( *, '("In Fortran:  line # = ", i2 )' ) n
      end
```

Code Example 6-44 Sharing I/O (C++ Code)

MixIOmain.cc

```
#include <stdio.h>

extern "C" {
    void mixio_( int& );
    void f_init();
    void f_exit();
};

main ( ) {
    f_init();
    int m= -9;

    for( int i= 0; i < 5; ++i ) {
    ++m;
    if( m == 2  ||  m == 4 ) {
        printf( "In  C++  :  line # = %d\n", m );
    } else {
        mixio_( m );
    }
    }
    f_exit();
    return 0;
}
```

Compile and execute, with output.

```
% f77 -c -silent MixIO.f
% CC -c -w MixIOmain.cc
% CC MixIO.o MixIOmain.o -lM77 -lF77
% a.out
error: negative line #
In FORTRAN: line # = 1
In C: line # = 2
In FORTRAN: line # = 3
In C: line # = 4
In FORTRAN: line # = 5
```

With a C++ `main()` program, the following FORTRAN library routines may not work correctly: `signal()`, `getarg()`, `iargc()`

Alternate Returns

Your C++ program may need to use a FORTRAN subroutine that has nonstandard returns. To C++, such subroutines return an `int` (`INTEGER*4`). The return value specifies which alternate return to use. If the subroutine has no entry points with alternate return arguments, the returned value is undefined.

Code Example 6-45 returns one regular argument and two alternate returns.

Code Example 6-45 Alternate Returns

AltRet.f

```
      subroutine AltRet ( i, *, * )
      integer i, k
      i = 9
      write( *, * ) 'k:'
      read( *, * ) k
      if( k .eq. 10 ) return 1
      if( k .eq. 20 ) return 2
      return
      end
```

Code Example 6-45 Alternate Returns

AltRetmain.cc

```
#include <stdio.h>
#include <stdlib.h>

extern "C" int altret_ ( int& );

main ( ) {
    int k = 0;
    int m = altret_( k );
    printf( "%d %d\n", k, m );
    return 0;
}
```

C++ invokes the subroutine as a function.

Compile, link, and execute:

```
% f77 -c -silent AltRet.f
% CC -c AltRetmain.cc
% CC AltRet.o AltRetmain.o -lM77 -lF77
% a.out
k:
20
9 2
```

In this example, the C++ `main()` receives a 2 as the return value of the subroutine, because you typed in a `20`.

Programming Environment 7

The C++ compiler runs in a programming environment that consists of a number of tools.

Text Editing

There are several text editors available.

`vi`—The major text editor for source programs is `vi` (pronounced “vee-eye”), the visual display editor. `vi` has considerable power because it offers the capabilities of both a line and a screen editor. `It` also provides several commands specifically for editing programs, which you can use in the editor.

Two examples are:

- The `autoindent` option, which supplies white space at the beginning of a line
- The `showmatch` option, which shows matching parentheses

For more information, see the `vi`(1) man page

`textedit`—The `textedit` editor and other editors are available, including `ed` and `ex`.

`emacs`—For the `emacs` editor, and other editors not from Sun, see the Sun document, *Catalyst™, a Catalog of Third-Party Software and Hardware.*

7

Program Development

The following is a brief description of the development tools available. See the *Programming Utilities and Libraries* manual for details.

`lex`—Generates programs used in simple lexical analysis of text, solves problems by recognizing different strings of characters.

`yacc`—Takes a description of a syntax and generates a C function to parse the input stream according to that syntax.

`prof`—Produces an execution profile of the modules in a program.

`make`—Automatically maintains, updates, and regenerates related programs and files.

System V make—Describes a version of `make` that is compatible with older versions of the tool.

`sccs`—Controls access to shared files and keeps a history of changes made to a project.

`m4`—Processes the macro language.

`error`—Generates a source code listing with compiler diagnostics printed before the source line that causes the error.

`gprof`—Profiles by procedure.

`sbrowser` — This is a utility with window, icon, mouse, and pointer interface, that functions as a source code and call graph browser, finds occurrences of any symbol. It can find them in all source files, including header files.

`tcov`—Profiles by statement.

`vgrind`—Formats program sources.

Analyzer—Tunes program performance, including memory allocation.

FileMerge—Merges source files and coordinating source code changes with other developers.

MakeTool—Builds programs and browses makefiles.

Manager—Manages and coordinates other programming tools.

Debugging

The debugging tools are:

- `dbx`—An interactive symbolic debugger that understands C++ programs.
- `debugger`—A window, icon, mouse, and pointer (WIMP) interface to `dbx`.
- Thread Analyzer—A multithreaded code analyzer. Available with SPARCworks™/iMPact.

7

Code Samples

Here is an example C++ program illustrating C++ features. It consists of:

- A sample implementation of a string class
- A small program to test it

The string class is an example which, athough not full-featured, illustrates features that a real class must have. Code Example 0-1 is the header file `str.h` for the string class `string`.

Code Example 0-1 Header File `str.h`

```
// header file str.h for toy C++ strings package
#include <string.h> // for C library string functions
class ostream;  // so we can declare output of strings

class string {
public:
    string();
    string(char *);
    void append(char *);
    const char* str() const;
    string operator+(const string&) const;
    const string& operator=(const string&);
    const string& operator=(const char*);
    friend ostream& operator<<(ostream&, const string&);
    friend istream& operator>>(istream&, string&);

private:
    char *data;
```

Code Example 0-1 Header File `str.h` *(Continued)*

```
    size_t size;
};
inline string::string() { size = 0; data = NULL; }
inline const char* string::str() const { return data; }

ostream& operator<<(ostream&, const string&);
istream& operator>>(istream&, string&);
```

Code Example 0-2 is an implementation file `str.cc` of the string class functions.

Code Example 0-2 String Class File `str.cc`

```
****************** str.cc ******************
// implementation for toy C++ strings package

#include <iostream.h>
#include "str.h"

string::string(char *aStr)
{
    if (aStr == NULL)
        size = 0;
    else
        size = strlen(aStr);

    if (size == 0)
        data = NULL;
    else {
        data = new char [size+1];
        strcpy(data, aStr);
    }
}

void string::append(char *app)
{
    size_t appsize = app ? strlen(app) : 0;
    char *holder = new char [size + appsize + 1];

    if (size)
        strcpy(holder, data);
    else
        holder[0] = 0;
```

Code Example 0-2 String Class File `str.cc` *(Continued)*

```
****************** str.cc ******************
    if (app) {
        strcpy(&holder[size], app);
        size += appsize;
    }
    delete [] data;
    data = holder;
}

string string::operator+(const string& second) const
{
    string temp;
    temp.data = new char[size + second.size + 1];

    if (size)
        strcpy(temp.data, data);
    else
        temp.data[0] = 0;

    if (second.size)
        strcpy(&temp.data[size], second.data);
    temp.size = size + second.size;
    return temp;
}

const string& string::operator=(const string& second)
{
    if (this == &second)
        return *this;   // in case string = string

    delete [] data;

    if (second.size)  {
        data = new char[second.size+1];
        size = second.size;
        strcpy(data, second.data);
    }
    else {
        data = NULL;
        size = 0;
    }
    return *this;
}
```

Code Example 0-2 String Class File `str.cc` *(Continued)*

```
****************** str.cc ******************
const string& string::operator=(const char* str)
{
    delete [] data;

    if (str && str[0])  {
        size = strlen(str);
        data = new char[size+1];
        strcpy(data, str);
    }
    else {
        data = NULL;
        size = 0;
    }
    return *this;
}

ostream& operator<< (ostream& ostr, const string& output)
{
    return ostr << output.data;
}

istream& operator>> (istream& istr, string& input)
{
    const int maxline = 256;
    char holder[maxline];
    istr.get(holder, maxline, '\n');
    input = holder;
    return istr;
}
```

Index

K

L

M

N

O

P

R

S

T

U

V

W