C++ Programmer’s Guide

About This Guide

This guide describes how to use the Silicon Graphics C++ compiler environment. It discusses the two native C++ compilers for producing 32− and 64−bit objects, respectively. Some of the discussion will involve cfront, the C++ to C translator for the 5.2 (and earlier) versions of the operating system.

What This Guide Contains

This guide contains the following chapters:

- **Chapter 1, "Understanding the Silicon Graphics C++ Environment,"** describes the Silicon Graphics C++ environment and the issue of *cfront* compatibility.
- **Chapter 2, "Compiling, Linking, and Running C++ Programs,"** describes how to compile, link, and run C++ programs in the Silicon Graphics C++ environment.
- **Chapter 3, "Interfaces,"** contains information on linking C++ programs with libraries written in C, and vice versa.
- **Chapter 4, "Code Examples,"** provides code examples for *cfront* compatibility.
- **Chapter 5, "Common Pitfalls,"** discusses some common problems with C++ libraries and how to diagnose and solve them.
- **Chapter 6, "Using Templates,"** discusses how C++ templates are used in the Silicon Graphics C++ environment.

The glossary defines key terms for the Silicon Graphics C++ environment.

What You Should Know Before Reading This Guide

This guide assumes that you are familiar with C, C++, object−oriented programming, shared libraries, and dynamic loading.

Related Information

The following manuals provide reference information about the implementation of the C++ language from Silicon Graphics.

- *C++ Language System Overview*, which contains an overview of newer language features of C++. Most of the extensions take the form of removing restrictions on what can be expressed in C++.
- *C++ Language System Product Reference Manual*, which contains a general description of the C++ language.
- *C++ Language System Library*, which discusses the iostream support in the C++ library and describes a data−type complex that provides the basic facilities for using complex arithmetic in C++.

The following manuals provide related information that you may need when using the Silicon Graphics C++ environment.

- *Delta/C++ Programming Guide*, which discusses the dynamic class ("delta") extensions to the Silicon Graphics compiler environment.
- *MIPSpro Compiling, Debugging and Performance Tuning*, which discusses how to compile, debug, and tune the performance of programs written in the Silicon Graphics development environment (C, Fortran, and C++).

Conventions Used in This Guide

These are the typographical and graphic conventions used in this guide:

- **Bold**—Functions, option flags, and classes
- *Italics*—File names, button names, field names, variables, emphasis, glossary terms, and IRIX

commands

- Regular—Menu and window names, data types, keywords, and text
- "Quoted"—Menu choices
- `Fixed-width`—Code examples and command syntax
- **`Bold fixed-width`**—User input. Nonprinting **`<keys>`** are bracketed

Chapter 1

Understanding the Silicon Graphics C++ Environment

This chapter describes the Silicon Graphics C++ compiler environment and contains the following major sections:

- **"Silicon Graphics C++ Environment"** discusses the different Silicon Graphics C++ compilers for IRIX 6.0 and 5.2 (and later) systems.
- **"Comparing NCC and CC –64 to cfront"** compares the Silicon Graphics C++ to other C++ environments.
- **"Using the Compilers"** discusses the differences between the 32 and 64–bit version of the Silicon Graphics compilers, shows the command lines for the compilers, and gives some examples of typical command lines.
- **"cfront Compatibility"** discusses the restrictions on C++ code that are enforced by the Silicon Graphics C++ environment, but were not enforced by *cfront*.
- **"Debugging"** discusses the Silicon Graphics C++ debugging environment.

Silicon Graphics C++ Environment

The Silicon Graphics C++ environment is available in two variaties, targeted for IRIX 6.0 and 5.2 (and later) systems. The 6.0 32–bit compiler and the 5.2 compiler are functionally identical. See **Figure 1–1** for details.

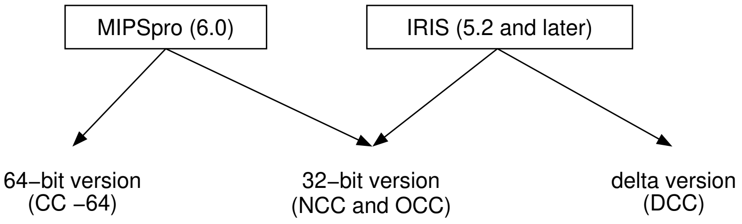

Figure 1–1 Silicon Graphics C++ Environment

IRIX 6.0 Compilers

As shown in **Figure 1–1** , there is a 32– and a 64–bit version of the C++ compiler for the IRIX 6.0 operating system. For completeness and backwards–compatibility, the old Silicon Graphics C++ compiler (*OCC*), based on *cfront*, is still made available. The supported C++ compilers for the 6.0 system are listed below:

CC –64	64 bit native C++ compiler.
NCC	32 bit native C++ compiler.
OCC	32–bit C++ compiler, based on C++ to C translation using *cfront*.

IRIX 5.2 (and Later) Compilers

There are also two versions of the C++ compiler for the 5.2 operating system: *NCC* and *DCC*. Again, for completeness and backwards–compatibility, the old Silicon Graphics C++ compiler (*OCC*), based on *cfront*, is still made available. The supported C++ compilers for the 5.2 system are listed below:

NCC	32 bit native C++ compiler. Functionally identical to 6.0 *NCC*.

DCC	Delta/C++ compiler, available as part of the Pro C++ package. The delta feature enables you to use dynamic classes, to minimize the need for recompilation upon changing a class, thereby greatly increasing productivity.
OCC	32–bit C++ compiler, based on C++ to C translation using *cfront*.

For complete details on *cfront* compatibility, see **"cfront Compatibility"** . For further details on the commands themselves, refer to the *CC(1)*, *NCC(1)*, *OCC(1)*, and *DCC(1)* reference pages.

Comparing NCC and CC –64 to cfront

There are a number of advantages to using the new compilers instead of *cfront*.

Native Compiling

NCC and *CC –64* are native compilers that are drop–in replacements for *cfront*. This is a major advantage, as *cfront* first translates C++ code to C, and then compiles the C code.

For instance, *OCC* (previously *CC*) runs *cpp*, then *cfront* (the C++ front end) on the C++ source to produce C source code. Then the *cc* command is invoked on the C code, which invokes *cfe* (the C front end). *NCC* just runs *edgcpfe*, the C++ front end, which does its own preprocessing.

Preprocessing, which is normally invoked as a separate process by *OCC*, is built into the NCC front end (*edgcpfe*). This eliminates the overhead of an entire preprocessing step and the cost of launching a separate process.

In addition, debugger support is much easier, since a native compiler can generate a richer set of symbolic debugger information than *cfront*.

Simplifying Template Instantiation

Template instantiation in current C++ systems is frequently laborious. For example, *cfront* saves information about each file it compiles in a special directory and instantiates nothing during normal compilations. At link time, it looks for entities that are referenced but not defined and whose names indicate that they are template entities. *cfront* then consults the special directory to find the file containing the source for the entity and compiles the source to generate an object file containing object code for that entity. This object code for instantiated objects is then combined with the "normal" object code in the link step.

NCC and *CC –64* instantiate template functions within your source files. For initial compilations, this makes no difference. For subsequent rebuilds, this reduces the time required to instantiate templates during the link phase and may prevent additional compilations, significantly improving compile–time performance. (*cfront* builds each template function in a separate object file, and the time required to compile each object file is significant.)

Improving cfront–Compiled Code

The compilers diagnoses several non–standard constructs quietly accepted by *cfront*, and fixes many *cfront* defects. Use *NCC* to avoid *cfront* compiler bugs and for stricter adherence to C++ standards.

Note: Exception handling is not fully supported in *NCC* and *CC –64*. *NCC* will optimize to *–O2* only. (*–O2* and *–O* are equivalent.) *OCC* supports no exception handling.

Using the Compilers

This section discusses how to use the Silicon Graphics compilers to compile your C++ programs. It describes the differences between the 64– and 32–bit versions of the compiler, describes the *CC –64*, *NCC*, and *OCC* command lines (and some of the more commonly used options), and contains some examples.

The default compiler is *NCC*. If you use *CC* with no options (*–64*, *–32*, or *–use_cfront*), you invoke *NCC*. If you use *CC* with options supported by *OCC* but not supported by *NCC*, or you use *CC* with the

−use_cfront option, you invoke *OCC*.

64− Versus 32−Bit Compilation

CC −64 and *NCC* are both native compilers that are based on the same front end. The *CC −64* front end is *fecc*, which has 64−bit pointers, addresses, and long ints. The *NCC* front end is *edgpcfe*, a front end with 32−bit pointers, addresses, and long ints.

Note: 64−bit objects are incompatible with 32−bit objects, and they cannot be linked together.

- Specify the *−64* option on the IRIX 6.0 command line to compile source files for 64−bit objects.
- Specify the *−32* option on the IRIX 6.0 command line to compile source files for 32−bit objects.

A compilation on an IRIX 5.3 system always produces 32−bit objects. The compiler back−end (optimizer and code generator) is different in *−32* and *−64* modes.

Refer to *MIPSpro Compiling, Debugging, and Performance Tuning* for a more complete discussion on how to set up the IRIX environment for *−32* versus *−64* compilers. Refer to the MIPSpro Porting and Transition Guide for further information on *−64* compilers.

Note: The template instantiation options and the warning number, used in the *−woff* option, is different between *CC −64* and *NCC*.

CC −64 Command Line

The command line for compiling in 64−bit mode is shown below.

```
CC −64 [ option ] . . . file . . .
```

CC −64 compiles with many of the same options as cc(1). The *−64* option may occur anywhere on the command line.See the *CC(1)* reference page for more information.

Note: *cfront* compatibility mode is disabled by default when you compile in 64−bit mode.

For complete information on all the options available with *CC −64*, see the *CC(1)* reference page.

NCC Command Line

The command line for compiling in 32−bit mode is shown below.

```
NCC [ option ] . . . file . . .
```

You may also use the following command line IRIX 6.0 systems:

```
CC −32 [ option ] . . . file . . .
```

The *−32* option may occur anywhere on the command line.

Note: *cfront* compatibility mode is enabled by default when you compile in 32−bit mode.

For complete information on all the options available with *NCC*, see the *NCC(1)* reference page.

OCC Command Line

The command line for compiling with *cfront* is shown below.

```
OCC [ option ] . . . file . . .
```

You may also use the following command line on:

```
CC −use_cfront [ option ] . . . file . . .
```

For complete information on all the options available with *OCC*, see the *OCC(1)* reference page.

Sample Command Lines

Some typical C++ compiler command lines are given below.

- To suppress the loading phase of your compilation and compile only one program, the command line is the following:

  ```
  CC −c program
  ```

- To compile with full warning about questionable constructs, the command line is the following:

  ```
  CC −fullwarn program1 program2 . . .
  ```

- To compile with warning messages off, the command line is the following:

  ```
  CC −w program1 program2 . . .
  ```

- To compile in 64−bit mode with *cfront* compatibility enabled, the command line is the following:

  ```
  CC −64 −cfront program1 program2 . . .
  ```

- To compile in 32−bit mode with cfront compatibility disabled, the command line is the following:

  ```
  CC −32 +p program1 program2 . . .
  ```

cfront Compatibility

The Silicon Graphics compilers (with the exception of *OCC*) force you to adhere to C++ code standards more strictly than *cfront* does. Code that you compiled successfully with *cfront* may not compile under the Silicon Graphics C++ environment, even in *cfront* compatibility mode. You must compile with *OCC* to get exact cfront compatibility. This section discusses *cfront* compatibility restrictions when compiling with *NCC* or *CC −64 −cfront*.

Compatibility Restrictions

In some cases, the Silicon Graphics C++ compilers are not backwards−compatible with *cfront* because *cfront* has defects, behaves in a non−deterministic manner, or fails to adhere to the standard. The C++ incompatibilities that *cfront* ignores but the Silicon Graphics compilers catch are listed below:

- You must have an explicit declaration of constructors or destructors in the class if there is an explicit definition of it outside the class. See **Example 4−1** for an example.
- You must not use the same identifier for more than one formal argument in a function definition. See **Example 4−2** for an example.
- The compiler cannot find a constructor that can accept an array, even though one exists that accepts a reference to an array. See **Example 4−3** for an example.
- You may not delete a pointer to a const. See **Example 4−4** for an example.
- If a C++−style (//) comment line is terminated with a backslash, the Silicon Graphics compiler will (correctly) continue the comment line into the next source line. (*cfront*, which uses the standard UNIX *cpp*, incorrectly terminates the comment at the end of the line.) See **Example 4−5** for an example.
- You may not pass a pointer to volatile data to a function that is expecting a pointer to non−volatile data. See **Example 4−6** for an example.
- The Silicon Graphics compiler does not disambiguate between overloaded functions with a char* and long parameter, respectively, when called with an expression that is a 0 cast to a char type. See **Example 4−7** for an example.
- You may not use redundant type specifiers. See **Example 4−8** for an example.
- When in a conditional expression, the Silicon Graphics compiler does not convert a pointer to a class to an accessible base class of that class. See **Example 4−9** for an example.
- You may not assign a 0 to a pointer, if the 0 is the right−expression of a comma operator. See **Example 4−10** for an example.
- The Silicon Graphics compiler mangles member functions declared as extern "C" differently from *cfront*. *NCC* does not strip the type signature when you are building the mangled name. If you try to do so, you will see the following warning:

```
Mangling of classes within an extern "C" block does not match cfront name
 mangling.
```

You may not be able to link code containing a call to such a function with code containing the definition of the function that was compiled with *cfront*.

- *cfront* incorrectly strips the "signed" keyword whenever it encounters a signed declaration (for example, **`signed char`**), and mangles the resulting name accordingly. The generated object file contains a different name for member functions with parameters having this type.

  The Silicon Graphics compiler does not do this, so if the definition of such a member function is compiled with *cfront*, and a call to it is compiled by the Silicon Graphics compiler, the code will not link. If you try to do so, you will see the following warning:

  ```
  Mangling of signed character does not match cfront name mangling.
  ```

  You must compile both the definition and the call with the Silicon Graphics compiler. If you do not have access to the source file defining the function, modify the declaration of the function in its header file by removing the signed keyword, or replacing it with an unsigned keyword.

  cfront also removes declarations that include "const".

For More Information

For examples of code that was valid under the *cfront* environment but is invalid in the Silicon Graphics C++ environment, see **"cfront Compatibility Examples"** .

C++ Libraries

By default, all C++ programs link with the standard library *libC.so*. This library contains all the *iostream* library functions, as well as the C++ storage allocation functions *::new* and *::delete*.

Silicon Graphics also provides the complex arithmetic library *libcomplex.a*. If you want to use this package you must explicitly link with this library. For example,

```
CC complexapp.c++ –lcomplex
```

See the *C++ Language System Library* for more information on the *complex* and *iostream* libraries.

Debugging

You can debug your C++ programs with the *dbx* debugger. For complete information on dbx, see the chapter on "Using the Debugger" in the *MIPSpro Compiling, Debugging and Performance Tuning Guide*.

Chapter 2

Compiling, Linking, and Running C++ Programs

This chapter contains the following major sections:

- **"Compiling and Linking"** describes the compilation environment and how to compile and link C++ programs. Some examples show how to create separate linkable objects in C++, C, Fortran, or other languages, and how to link them into an executable program.
- **"Translator Options"** describes the command line options that can be provided to the C++ translator.
- **"Object File Tools"** briefly summarizes the capabilities of the tools that provide symbol and other information on object files.

Compiling and Linking

This section discusses Silicon Graphics C++ compiling and linking.

Translators and Drivers

Programs called *drivers* invoke the major components of the compiler system. Those components, their functions, and their place in the compilation process are discussed in the following sections. The *CC(1)* command invokes the driver that controls compilation of your C++ source files. The syntax is as follows:

```
CC [options] filename.C [options] [filename2.C ...]
```

where:

CC	invokes the various processing phases that translate, compile, optimize, assemble, and compile−time link the program.
options	represents the driver options, which give instructions to the processing phases. Options can appear anywhere in the command line. The options interpreted by *CC* are discussed in **"Translator Options"** in this chapter.
filename.C	is the name of the file that contains the C++ source statements. The file name must end with one of the following acceptable suffixes: *.C*, *.c++*, *.c*, *.cc*, or *.cxx*.

Compilation

The compilation process shown in **Figure 2−1** is that of the C++ source file *foo.C*, as it would be compiled by this command line:

```
CC −o foo foo.C
```

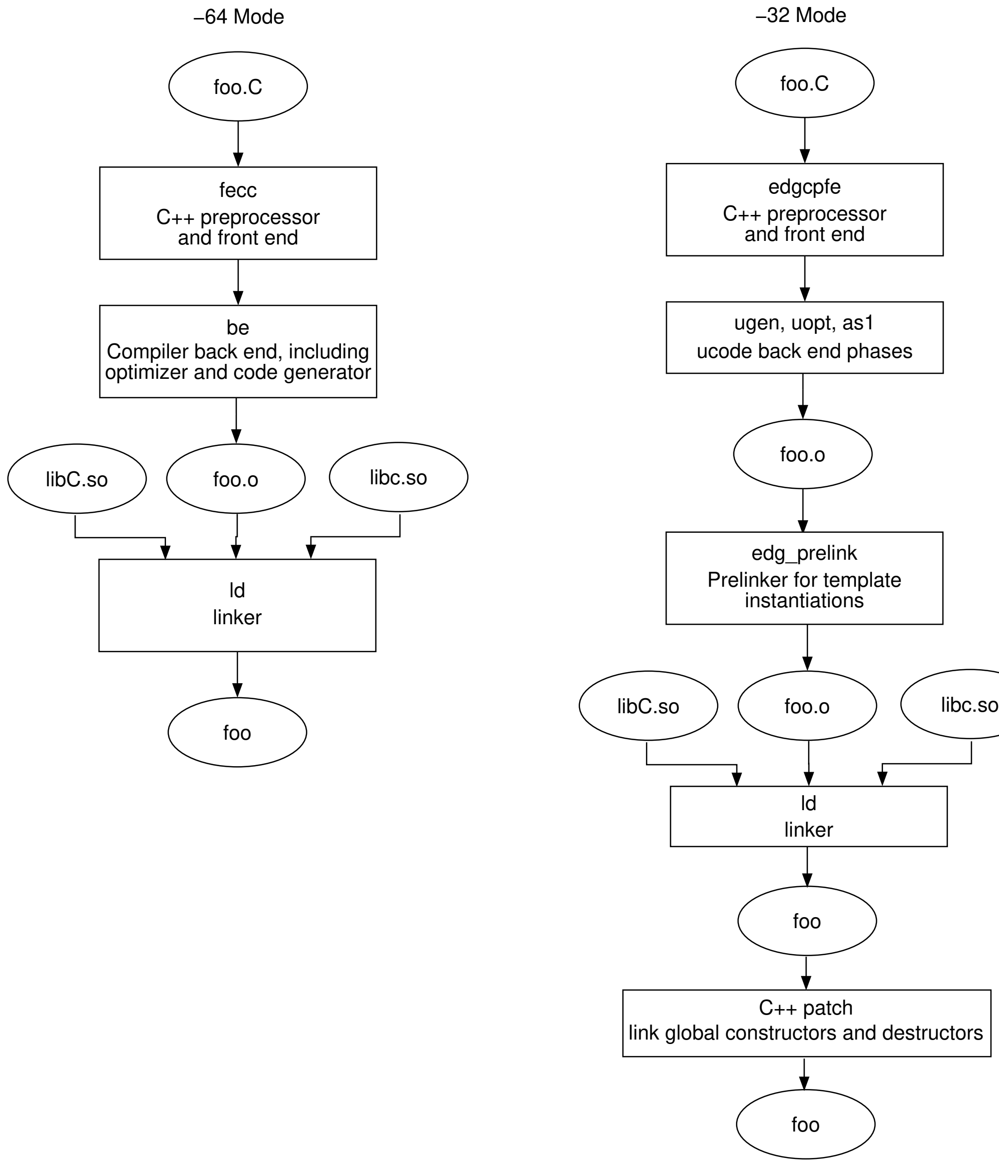

Figure 2–1 The Compilation Process

The following steps further describe the stages of compilation:

1. You invoke *CC* on the source file, which ends with the suffix *.C*. The other acceptable suffixes are *.c++*, *.c*, *.cc*, and *.cpp*.
2. The source file then passes through the C++ preprocessor, which is built into the C++ front end (*fecc* , *edgpcfe*).
3. The complete source is then processed by the C++ front end (*fecc* or *edgcpfe*), which produces an intermediate representative from a syntactic and semantic analysis of the source.

   This stage may also produce a prelink (*.ii*) file, which contains information about template instantiations.
4. The back end (*be* in *–64* mode) generates optimized object code (*foo.o*).
5. If you want to stop the compilation at this phase, and produce object code suitable for later linking, use the following command:

   ```
   CC −c foo.c
   ```

   The object file *foo.o* is the result.
6. *edg_prelink* processes the *.ii* files associated with the objects that will be linked together. It then

recompiles sources to force template instantiation.

7. The object files are sent to the linker *ld*(1), which links the standard C++ library *libC.so* and the standard C library *libc.so* to the object file *foo.o* and to any other object files that need to be linked to produce the executable.
8. In *–32* mode only, the executable object is sent to *c++patch*, which links it with global constructors and destructors. If global objects with constructors or destructors are present, the constructors need to be called at run time before the objects are used, and the destructors need to be called when the program exits. *c++patch* modifies the executable (*a.out*) to insure that these constructors and destructors get called.

Multi–Language Programs

C++ programs can be compiled and linked with programs written in other languages, such as C, Fortran, and Pascal. When your application has two or more source programs written in different languages, you should compile each program module separately with the appropriate driver. Then you can link them in a separate step. You can create objects suitable for linking by specifying the *-c* option. For example:

```
CC -c main.c++
f77 -c module1.f
cc -c module2.c
```

The various compilers would produce three object files: *main.o*, *module1.o*, and *module2.o*. Since the main module is written in C++, you should use the *CC* command to link. Except for C, you must explicitly specify the link libraries for the other languages with the *-l* options. For example, to link the C++ main module with the Fortran submodule, you would use the following command:

```
CC -o almostall main.o module1.o -lF77 -lI77 -lisam -lm
```

For further information on libraries for other languages, see the appropriate programmer's guides. See also **Chapter 3, "Interfaces,"** in this guide for a discussion of other topics important to writing multi–language programs.

For more information on C++ libraries, see **"C++ Libraries"** .

Translator Options

This section contains a summary of the most important *CC* translator options. See the reference page for *CC*(1) for a complete description of all the options. See *ld*(1) for a description of the linker options, and *cc* (1) for a description of the options interpreted by the standard C compiler. See also the information in *MIPSpro Compiling, Debugging and Performance Tuning*.

-E	Run only *cpp*(1) on the C++ source files and send the result to standard output. This option is useful, for example, if you want to see exactly which files were included in your compilation.
-c	Produce object files only, suppressing the link phase.

-o output	Name the final output file *output*. For example,
	`CC -o foo foo.C`
	produces an executable called *foo* instead of the default *a.out*. The command is shown below.
	`CC -c -o bar.o foo.C`
	produces an object file called *bar.o* instead of the default *foo.o*.

-n (*–32* mode), *–show* (*–64* mode)	Print commands generated by *CC* but do not execute them.
+v (*–32* mode), *–show* (*–64* mode)	Print commands as they are executed. Short for *verbose* output.
+d (*–32* mode), *–noinline* (*–64* mode)	Do not attempt inline substitution for calls to functions declared as *inline*.
+w (*–32* mode), *–fullwarn* (*–64* mode)	

	warnings only about constructs that are almost certainly problems.
+p (*–32* mode), *–anach* (*–64* mode)	Disallow all anachronistic constructs. Ordinarily, the translator warns about anachronistic constructs. Under *+p* (for pure), the translator will not compile code containing anachronistic constructs, such as "assignment to this". See the *USL C++ Language System Product Reference* for a list of anachronisms. In *–32* mode, *+p* also disables *cfront* compatibility mode, enforcing a stricter, more standard language definition. In *–64* mode, the stricter definition is the default.
–use_cfront (*–32* mode)	Use *OCC* instead of *NCC*.
–cfront (*–64* mode)	Compile in cfront compatibility mode. This is the default in *–32* mode.
-nocpp	Skip the preprocessing stage.

Object File Tools

For information on the object file tools available to you, consult the *IRIS–4D Series Compiler Guide*. The following tools are of special interest to the C++ programmer:

nm — The *nm* tool can be used to print symbol table information for object files and archive files.

c++filt — This C++–specific tool translates the internally coded (mangled) names generated by the C++ translator into names more easily recognized by the programmer. You can, for example, pipe the output of *stdump* or *nm* into *c++filt*. *c++filt* is installed in the directory */usr/lib/c++*. For example,

```
nm a.out | /usr/lib/c++/c++filt
```

libmangle.a — The library */usr/lib/c++/libmangle.a* provides a function *demangle(char *)* that you can invoke from your own program to give you a readable form of a mangled name. This is useful if you want to write your own tool for processing the output of *nm*, for example. You need to include the declaration

```
char * demangle(char *);
```

in your program, and link with the library

```
/usr/lib/c++/libmangle.a.
```

size — The *size* tool prints information about the text, rdata, data, sdata, bss, and sbss sections of the specific object or archive file. The contents and format of section data are described in Chapter 10 of the *Assembly Language Programming Guide*.

Chapter 3

Interfaces

This chapter contains the following major sections:

- **"Using Other Language Libraries from C++ Programs"** describes how to link the libraries from other languages into your C++ program.
- **"C Function Declarations in C++ Programs"** describes how to use declarations in the Silicon Graphics C++ environment.
- **"Using C++ Libraries From C Programs"** how to call C++ libraries from C programs.

Using Other Language Libraries from C++ Programs

Most C++ programmers want to be able to link with object files and libraries written in languages other than C++, especially C. In order to do so, you must include in your programs declarations for the functions you wish to call. In most cases you can do this by simply including appropriate header files with the *#include* directive. For the standard C header files supplied by Silicon Graphics, using *#include* is all you need to do. For example, if you are going to use C standard I/O, you must include the following line in your source or header file:

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

To use the Graphics Library™, you must include the following line in your source or header file:

```
#include <gl/gl.h>
```

C Function Declarations in C++ Programs

If you want to call C functions from a C++ program, either directly or by file inclusion, you must be sure the C++ program contains correctly prototyped declarations for the functions, and that the function declarations are recognizable by the C++ compiler as declaring functions whose definitions are in C. These steps are necessary because C++ normally encodes (mangles) function names to support overloading. The real name of a function declared in a C++ program as *void printf(char*, ...)*, for example, is *printf__FPce*. The *printf* function in *libc.a*, however, is just called *printf*.

To allow a C++ program to call functions written in C, C++ provides linkage specifications. To use the standard *printf* function, for example, you could write the following code:

```
extern "C" {
     void printf(char *, ...);
}
```

You can include this code in the C++ source file that calls *printf*, or in a header file that is included by the source file. The use of *extern "C"* tells the translator that the function linkage should be done according to the conventions used by the C programming language, in particular, with no mangling. A function name declared in this way is said to have C linkage.

If you want to adapt an existing C header file or create a header file of your own containing C function declarations, and you want to be able to include it in either C or C++ programs, you can use the fact that the symbol *__cplusplus* (with two underscores preceding it) is always defined for C++ compilations and is always undefined otherwise. Thus, you can enclose C function declarations with the following code:

```
#ifdef __cplusplus
extern "C" {
#endif
```

and

```
#ifdef __cplusplus
}
#endif
```

This scheme is used to create the Silicon Graphics C header files.

Using C++ Libraries From C Programs

You may want to use libraries created by C++ from programs written in C. One way to do this is to create interface functions written in C++ but declared as having C linkage so they can be called from C programs. Suppose, for example, that you have a C++ library that implements a symbol table class. It might have a class declaration that looks like the sample from *table.h* below:

```
class symtab {
   ...
public:
   symtab();
   void add_name(char *);
   int lookup(char *);
   ...
};
```

You might create an interface header file (called *interface.h*, for example) that looks like the sample below:

```
extern "C" {
   void init_table(void);
   void add_name(char *);
   int lookup(char *);
   ...
}
```

You would also create an interface program (called *interface.c*, for example) that looks like the sample below:

```
#include "table.h" // class declaration for symtab
#include "interface.h"
static symtab * stab;
void init_table(void)
{
   stab = new symtab;
}

void add_name(char *name)
{
   stab->add_name(name);
}

int lookup(char * name)
{
   return stab->lookup(name);
}
```

A problem often encountered in using C++ libraries from C Programs is that global objects defined in the library are not initialized if the main program is not in C++. To overcome this problem, link your program using *CC* instead of *cc*. Linking using *CC* ensures that the linked executable will have its C++ global objects correctly initialized on program startup.

If your program (or DSO) includes C++ object files that have global objects that need to be constructed at program startup, link your program (or DSO) using *CC*.

Chapter 4

Code Examples

This chapter contains the following major sections:

- **"cfront Compatibility Examples"** provides you with examples of code that compiled with *cfront*, but is invalid in the Silicon Graphics C++ environment.

cfront Compatibility Examples

This section contains examples of *cfront*–compatible code that are illegal in the Silicon Graphics C++ environment. (For a complete description of the *cfront* compatibility issues, see **"cfront Compatibility"** .) The examples covered are listed below:

- Explicitly declaring member functions
- Using the same identifier for multiple arguments
- Finding a constructor that accepts an array
- Deleting a pointer to a const
- Terminating comment lines with a backslash
- Passing a pointer to volatile data
- Disambiguating between a char* and a long
- Rejecting redundant type specifiers

- Converting a pointer to a class to an accessible base class
- Assigning a 0 to a pointer

Explicitly Declaring Member Functions

You must have an explicit declaration of a member function in a class if there is an explicit definition of it outside the class. *cfront* allows this, but *DCC* displays the following error message:

```
error(414): defining an implicitly declared member function is not allowed
 alpha::alpha( const alpha &other )
      ^
```

For example, the code in **Example 4–1** will not compile.

Example 4–1 Explicitly Declaring Member Functions

```
class alpha
{
public:
 virtual void func(); // Needed to get implicit constructor
 long member; // No constructor, so not initialized
};

alpha::alpha( const alpha &other )
{
 member = other.member + 1000;
}
```

You must explicitly declare the missing member function and provide the default constructor.

Using the Same Identifier for Multiple Arguments

If you use the same identifier for more than one formal argument in a function definition, both *cfront* and *DCC* issue the following error:

```
error(100): duplicate parameter name
 int funcdef( int arg, char arg ) // Right error 100.
          ^
```

However, you may not use the same identifier for more than one formal argument in a function declaration. *cfront* allows this, but *DCC* issues the following error:

```
error(100): duplicate parameter name
extern void memfunc( int arg, float *arg ); //Wrong error 100
          ^
```

For example, the code in **Example 4–2** will not compile.

Example 4–2 Using the Same Identifier for Multiple Arguments

```
struct S
{
 void memfunc( int arg, float* arg );
};
```

You must use different names, or no names at all.

Finding a Constructor That Accepts an Array

The compiler cannot find a constructor that can accept an array, even though one exists that accepts a reference to an array. If you have such a piece of code, the following error message is displayed:

```
error (496): no suitable constructor exists to convert from
```

```
     "float *" to "Foo"
   X = A;  // Error
       ^
```

For example, the code in **Example 4–3** will not compile.

Example 4–3 Finding a Constructor That Accepts an Array

```
typedef float Array[4];  // No error if this is a float*

class Foo {
public:
 Foo(){}
    Foo( Array& m );        // No error if this not a reference
};

void g(Array& ) {}

void f()
{
    Array A;
    Foo   X;

    X = A;                  // Error
    X = Foo(A);             // Work-around
    g( A );                 // Works OK
}
```

As shown, you must explicitly invoke the constructor.

Deleting a Pointer to a const

You may not delete a pointer to a const. (If the pointer is to a type other than the class itself, *cfront* also issues an error message.) If you do so, the following error message is displayed:

```
error(352): a pointer to const may not be deleted
 delete ptr; // Error 352.
   ^
```

For example, the code in **Example 4–4** will not compile.

Example 4–4 Deleting a Pointer to a const

```
class Foo
{
public:
 Foo();
 Foo( int );
 ~Foo();
private:
 int   stuff;
 const Foo* ptr;
};
Foo::~Foo()
{
 delete ptr; // Error 352.
}
```

You must cast the pointer to a non–const type.

Terminating Comment Lines With a Backslash

If you use a C++−style (//) comment line terminated with a backslash, *DCC* will (correctly) continue the comment line into the next source line. *CC* (which uses the UNIX® standard *cpp*) incorrectly terminates the comment at the end of the line. This may cause hard−to−find bugs.

In **Example 4−5** , *DCC* considers the two lines following the // comment to be part of the comment, while *cfront* does not.

Example 4−5 Terminating Comment Lines With a Backslash

```
#include <stdio.h>

int macro() { return 0; }

// Continued comment...............\
#define macro() \
 (printf( "FAIL: executed a comment!\n" ), 1 )

int main()
{
 return macro();
}
```

cfront (incorrectly) calls the macro and *DCC* calls the function.

You must delete the backslash at the end of the comment line.

Passing a Pointer to Volatile Data

You may not pass a pointer to volatile data to a function that is expecting a pointer to non−volatile data. If you do so, the following error message is displayed:

```
error(252): argument of type "volatile mytype *" is
 incompatible with parameter of type "mytype *"
 f1( vol ); // Error.
   ^
```

For example, the code in **Example 4−6** will not compile.

Example 4−6 Passing a Pointer to Volatile Data

```
typedef struct
{
 long i;
 long j;
} mytype;

void f1( mytype* );
void f2( int* );

volatile mytype *vol;
volatile int *pvi;

int main()
{
 f1( vol ); // Error.
 f2( pvi ); // Error.
 return 0;
}
```

One solution is to cast the actual expression to the type expected, though that may result in incorrect code, as the data is no longer treated as volatile. A better solution is to rewrite the function to accept volatile data.

Disambiguating Between a char* and a long

You must make a distinction between a char* and a long when converting an unsigned char having a zero value. If you do not, the following error message is displayed.

```
error(390): more than one instance of constructor
 "UndoEvent::UndoEvent" matches the argument list:
 function "UndoEvent::UndoEvent(long, char *)"
 function "UndoEvent::UndoEvent(long, long)"
 UndoEvent* x = new UndoEvent( 906, (tuBool)False );
```

For example, the code in **Example 4–7** will not compile.

Example 4–7 Disambiguating Between a char* and a long

```
class UndoEvent {
public:
 UndoEvent( long, char * ){}
 UndoEvent( long, long ){}
};

typedef unsigned char tuBool;
#define True 1
#define False 0

int main()
{
 UndoEvent* x = new UndoEvent( 906, False ); // Error 390.
 return 0;
}
```

You must use a cast to disambiguate which constructor is to be called.

Rejecting Redundant Type Specifiers

You may not use redundant type specifiers. If you do so, the following error message is displayed:

```
error(177): invalid combination of type specifiers
    unsigned unsigned int x; // Error
             ^
```

For example, the code in **Example 4–8** will not compile.

Example 4–8 Rejecting Redundant Type Specifiers

```
typedef const int Int;
. . .
const Int x;            // Error
```

You must delete the redundant type specifier.

Converting a Pointer to a Class to an Accessible Base Class

You cannot convert a pointer to a class to an accessible base class of that class with *DCC*. If you try to do so, the following error message is displayed:

```
error(135): operand types are incompatible ("D1 *" and "D2 *")
 bp = i ? dp1 : dp2; // Error.
      ^
```

For example, the code in **Example 4–9** will not compile.

Example 4–9 Converting a Pointer to a Class to an Accessible Base Class

```
struct base { };

struct D1 : public base { };
struct D2 : public base { };

int i =     906;
base* bp = (base*)  0;
D1* dp1 = (D1*)   1;
D2* dp2 = (D2*)   2;

void main(void)
{
 bp = i ? dp1 : dp2; // Error.
 bp = i ? (base*)dp1 : (base*)dp2; // Work-around.
}
```

You must cast both the second and third expressions to the type of the expression assigned.

Assigning a 0 to a Pointer

You cannot assign a 0 to a pointer if the 0 is the right expression of a comma operator. If you do, the following error message is displayed:

```
error(611): a value of type "int" cannot be assigned to
 an entity of type "int *"
 p = CAST ( 0, 0 ); // Error.
  ^
```

For example, the code in **Example 4–10** will not compile.

Example 4–10 Assigning a 0 to a Pointer

```
#ifdef WORKAROUND
#define CAST (int*)
#else
#define CAST /* nothing */
#endif

void main()
{
 int* p;
 p = CAST ( 0, 0 ); // Error.
}
```

You must cast the entire comma expression to the type of the pointer being assigned. You can do this by setting the macro WORKAROUND.

Chapter 5

Common Pitfalls

This chapter contains the following major sections:

- **"Problems Involving C Linkage"** discusses some problems you may encounter when you link your C++ programs to the C libraries.
- **"Problems With Order of Specification of Libraries"** discusses some problems you may encounter when you order the libraries on the command line.

Problems Involving C Linkage

One of the most common problems you may encounter occurs when you link your C++ programs to C code (such as C libraries). This section contains many of the most typical problems you run into in that situation.

- Unexpected undefined symbols. You may see the following error message from the link−editor:

  ```
  Unresolved:  foo(int, char*)
  ```

  The presence of the prototype in this message indicates that this is a C++ function. Frequently this means not that the function *foo* is undefined, but that it is defined in a C object file or library, and the C++ declaration is missing an extern "C" linkage specification.
- Inconsistent Linkage. You may see the following error message from the C++ front end (*fecc*):

  ```
  "afile.c", line 37: error (1311): linkage specification is incomplete with
   previous foo (declared at line 17)
  ```

  This means that two declarations for *foo()* were found with the same prototype, the first outside an extern "C" specification and the second inside. For example, you may have the following code, all in one compilation unit:

  ```
  void foo(int, char*);
  ......
  extern "C" {  void foo(int, char*);  }
  ```

  Frequently these two declarations come from different header files.
- A "Two ... with c linkage" error. For example, you may see the following error message from the C++ front end (*edgcpfe*):

  ```
  "afile.C", line 37: error (3419): more than one instance of overloaded
   function "foo" has "C" linkage.
  ```

  This indicates that two declarations for *foo()* were found within extern "C" specifications but with different prototypes. Typically this happens when a function is declared in two header files with the wrong prototype in one of them, or when a function already declared in an included header file is redeclared incorrectly.

Problems With Order of Specification of Libraries

This section covers some typical problems you may encounter when you specify the order of your libraries.

- Inability to use the Silicon Graphics fast malloc routines, *malloc(3x)*.

  A related problem occurs with a command such as the following:

  ```
  CC foo.c++ −lmalloc
  ```

  The command mysteriously fails to use the "fast" *libmalloc.a* versions of *malloc()* and *free()*. Here again the order of libraries is

  ```
  −lmalloc −lC −lc
  ```

  At the time *ld* processes *libmalloc.a*, there are no undefined references to *malloc()* and *free()* (unless explicitly referenced from *foo.c++*). Only when *new* and *delete* are picked up from *libC.a* are *malloc()* and *free()* required, and then it is too late: their references have already been resolved from *libc.a* instead of *libmalloc.a*. Again, once the problem is recognized the solution is easy. Just change the command to the following:

  ```
  CC foo.c++ −lC −lmalloc
  ```

- Mixing *stdio* and *iostreams*.

  If you mix *iostream* output using *cout* with *stdio* output using *printf*, you may see unexpected results if you are not careful about flushing the output buffers. For example, consider the following program:

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

```
main() {
    cout << "cout1\n";
    printf("printf1\n");
    cout << "cout2\n";
    printf("printf2\n");
   }
```

This code produces the following output:

```
printf1
printf2
cout1
cout2
```

This is because *cout* and *printf* use distinct buffers, and insertion of a newline into *cout* does not flush the buffer. To flush the buffer, you can insert the manipulator *flush* into the stream in the following way:

```
cout << "cout1\n" << flush;
```

You can also use the manipulator *endl* to insert a newline and flush as follows:

```
cout << "cout1" << endl;
```

On the other hand, consider the following program

```
main() {
    cout << "cout1 " << flush;
    printf("printf1 ");
    cout << "cout2 " << flush;
   }
```

This code produces the output

```
cout1 cout2 printf1
```

This is because the buffer for *printf* is not flushed until the program terminates. Here you need to call *fflush* after the call to *printf* as follows:

```
fflush(stdout);
```

This generates the following "expected" output

```
cout1 printf1 cout2
```

Chapter 6

Using Templates

This chapter contains the following major sections:

- **"NCC Template Instantiation"** describes how you perform template instantiation in the 32–bit Silicon Graphics C++ environment.
- **"CC –64 Template Instantiation"** describes how you perform template instantiation in the 64–bit Silicon Graphics C++ environment.
- **"Template Language Support"** describes how template language is supported in the Silicon Graphics C++ environment.

cfront template support is discussed in the chapter "Automatic Template Instantiation" in the *C++ Language System Overview*.

NCC Template Instantiation

This section describes the Delta/C++ implementation of templates. It compares the use of templates in the Delta/C++ environment to those of the Borland C++ and *cfront* compilers.

The instantiation of a class template is always done as soon as it is needed in a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (hereafter referred to as template entities) are not necessarily done immediately. The reasons for this are given below.

- You should have only one copy of each instantiated entity across all the object files that make up a program. (This applies to entities with external linkage.)
- You may write a specialization of a template entity. (For example, you can write a version of *Stack<int>*, or of just *Stack<int>::push*, that replaces the template−generated version. Often, this kind of specialization is a more efficient representation for a particular data type.) When compiling a reference to a template entity, the compiler does not know if a specialization for that entity will be provided in another compilation. The compiler cannot do the instantiation automatically in any source file that references it.
- You may not compile template functions that are not referenced. Such functions might contain semantic errors that would prevent them from being compiled. A reference to a template class should not automatically instantiate all the member functions of that class.

Note: Certain template entities are always instantiated when used (for example, inline functions).

If the compiler is responsible for doing all the instantiations automatically, it can do so only on a program−wide basis. The compiler cannot make decisions about instantiation of template entities until it has seen all the source files that make up a complete program.

By default, *NCC* performs automatic instantiation at link time. It is also possible for you to instantiate all necessary template entities at compile time using the *−ptused* option. See **"Explicit Instantiation"** for further details.

Automatic Instantiation

Automatic instantiation enables you to compile source files to object code, link them, run the resulting program, and never worry about how the necessary instantiations are done.

NCC requires that for each instantiation you have a normal, top−level, explicitly−compiled source file that contains both the definition of the template entity and any types required for the particular instantiation.

Meeting Instantiation Requirements

You can meet the instantiation requirements in several ways:

- You can have each header file that declares a template entity contain either the definition of the entity or another file that contains the definition.
- When the compiler sees a template declaration in a header file and discovers a need to instantiate that entity, you can give it permission to search for an associated definition file having the same base name and a different suffix. The compiler implicitly includes that file at the end of the compilation. This method allows most programs written using the *cfront* convention to be compiled. See **"Implicit Inclusion"** .
- You can make sure that the files that define template entities also have the definitions of all the available types, and add code or *pragmas* in those files to request instantiation of the entities there.

Delta/C++ Automatic Instantiation Method

1. The first time the source files of a program are compiled, no template entities are instantiated. However, the generated object files contain information about things that could have been instantiated in each compilation.

2. When the object files are linked, a program called the prelinker is run. It examines the object files, looking for references and definitions of template entities, and for the added information about entities that could be instantiated.

3. If the prelinker finds a reference to a template entity for which there is no definition anywhere in the set of object files, it looks for a file that indicates that it could instantiate that template entity. When it finds such a file, it assigns the instantiation to it. The set of instantiations assigned to a given file, say

abc.C, is recorded in an associated *.ii* file (for example, *abc.ii*). All *.ii* files are stored in a directory named *ii_files* created below your object file directory.

4. The prelinker then executes the compiler again to recompile each file for which the *.ii* file was changed. (The *.ii* file contains enough information to allow the prelinker to determine which options should be used to compile the same file.)
5. When the compiler compiles a file, it reads the *.ii* file for that file and obeys the instantiation requests therein. It produces a new object file containing the requested template entities (and all the other things that were already in the object file).
6. The prelinker repeats steps 3–5 until there are no more instantiations to be adjusted.
7. The object files are linked.

Details of Automatic Instantiation

Once the program has been linked correctly, the *.ii* files contain a complete set of instantiation assignments. From then on, whenever source files are recompiled, the compiler will consult the *.ii* files and do the indicated instantiations as it does the normal compilations. Except in cases where the set of required instantiations changes, the prelink step will find that all the necessary instantiations are present in the object files and that no instantiation assignment adjustments need be done. This is true even if the entire program is recompiled.

If you provide a specialization of a template entity somewhere in the program, the specialization will be seen as a definition by the prelinker. Since that definition satisfies whatever references there might be to that entity, the prelinker will see no need to request an instantiation of the entity. If the programmer adds a specialization to a program that has previously been compiled, the prelinker will notice that too and remove the assignment of the instantiation from the proper *.ii* file.

The *.ii* files should not, in general, require any manual intervention. The only exception is if the conditions below are all met.

- A definition is changed in such a way that some instantiation no longer compiles (it generates errors).
- A specialization is simultaneously added in another file
- The first file is recompiled before the specialization file and is generating errors.

The *.ii* file for the file generating the errors must be deleted manually to allow the prelinker to regenerate it.

If the prelinker changes an instantiation assignment, it will issue a message:

```
C++ prelinker: f__10A__pt__2_iFv assigned to file test.o
C++ prelinker: executing: usr/lib/DCC/edg-prelink -c test.c
```

The name in the message is the mangled name of the entity. These messages are printed if you use the *–ptv* option.

The automatic instantiation scheme can coexist with partial explicit control of instantiation by the programmer, through the use of pragmas or command–line specification of the instantiation mode.

The automatic instantiation mode can be disabled by using the *–no_prelink* option.

If automatic instantiation is turned off,

- the extra information about template entities that could be instantiated in a file is not put into the object file
- the *.ii* file is not updated with the command line
- the prelinker is not invoked

Implicit Inclusion

For the best results, you must include all the template implementation files in your source files. Since most *cfront* users do not do this, the compiler attempts to find unincluded template bodies automatically. For example, suppose that the following conditions are all true.

- template entity *ABC::f* is declared in file *xyz.h*
- an instantiation of *ABC::f* is required in a compilation

- no definition of *ABC::f* appears in the source code processed by the compilation

In this case, the compiler looks to see if the source file *xyz.n* exists. (By default, the list of suffixes tried for *n* is *.c*, *.C*, *.cpp*, *.CPP*, *.cxx*, *.CXX*, and *.cc*.) If so, the compiler processes it as if it were included at the end of the main source file.

Implicit inclusion works well alongside automatic instantiation, but the two are independent. They can be enabled or disabled independently, and implicit inclusion is still useful when automatic instantiation is not done. Implicit inclusion can be disabled with the *−no_auto_include* option.

Explicit Instantiation

NCC instantiates all templates at compile time if you use the *−ptused* option. The compiler produces larger object files because it stores duplicate instantiations in the object files. The duplicate copies are removed by the linker, and do not exist in the final executables.

The *NCC* template instantiation mechanism also correctly handles static data members when you use the *−ptused* option. Static data members that need to be dynamically initialized may be instantiated in multiple compilation units. However, the dynamic initialization takes place only once. This is implemented by using a flag which is set the first time a static data member is initialized. This flag prevents further attempts to initialize it dynamically.

The *−ptused* option is acceptable for most small− or medium−sized applications. There are some drawbacks listed below:

- Instantiating everything produces large object files.
- Although duplicate code is removed, the associated debug information is not removed, producing large executables.
- If you change a template body, you must recompile every file that contains an instantiation of this body. (The easiest way to do this is for you to use *make* in conjunction with the *−MDupdate* option. See the *DCC(1)* reference page and **"Limitations"** for more information.)
- If you plan on specializing a template function instantiation, you may have to set **`pragma do_not_instantiate`** if it is likely that the compiler−generated instantiation will contain syntax errors.
- Data is not removed, so there are multiple copies of static data members.

You can exercise finer control over exactly what is instantiated in each object file by using pragmas and command−line options.

Command Line Options for Template Instantiation

You may use command−line options to control the instantiation behavior of the compiler. These options are divided into sets of related options, as shown below. You use one option from each category; options from the same category are not used together. (For example, you do not use *−ptnone* in conjunction with *−ptused*.)

- *−ptnone* (the default), *−ptused*, and *−ptall*
- *−prelink* (the default) and *−no_prelink*
- *−auto_include*, *−no_auto_include*
- *−ptv*

The command line options are listed below.

−ptnone	causes none of the template entities to be instantiated. If automatic instantiation is on (in other words, *−prelink*), any template entities that the prelinker instructs the compiler to instantiate are instantiated.
−ptused	causes any template entities used in this compilation unit to be instantiated. This includes all static members that have template definitions. If you specify *−ptused*, automatic instantiation is turned off by default. If you enable automatic instantiation explicitly (with *−prelink*), any additional template entities that the prelinker instructs the compiler to instantiate are also instantiated.
−ptall	causes any template entities declared or referenced in the current compilation unit to be instantiated. For each fully instantiated template class, all its member functions and static data members are instantiated whether or not they are used. Nonmember template functions are instantiated even if the only reference was a declaration. If you use *−ptall*, automatic instantiation is turned off by default. If you

declaration. If you use *–ptall*, automatic instantiation is turned off by default. If you enable automatic instantiation explicitly (with *–prelink*), any additional template entities that the prelinker instructs the compiler to instantiate are also instantiated.

–prelink	instructs the compiler to output information from the object file and an associated *.ii* file to help the prelinker determine which files should be responsible for instantiating the various template entities referenced in a set of object files. When *–prelink* is on, the compiler reads an associated *.ii* file to determine if any template entities should be instantiated. When *–prelink* is on and a link is being performed, the driver calls a "template prelinker." If the prelinker detects missing template entities, they are assigned to files (by updating the associated *.ii* file), and the prelinker recompiles the necessary source files.
–no_prelink	instructs the compiler to not read a *.ii* file to determine which template entities should be instantiated. The compiler will not store any information in the object file about which template entities could be instantiated. This option also directs the driver not to invoke the template prelinker at link time. This is the default mode if *–ptused* or *–ptall* are specified.
–auto_include	instructs the compiler to implicitly include template definition files if such definitions are needed. (See **"Implicit Inclusion"** .)
–no_auto_include	disables implicit inclusion of template implementation files. (See **"Implicit Inclusion"** .)
–ptv	puts the template prelinker in verbose mode; when a template entity is assigned to a particular source file, the name of the template entity and source file is printed.

Note: In the case where a single file is compiled and linked, the compiler uses the *–ptused* option to suppress automatic instantiation.

Command Line Instantiation Examples

This section provides you with combinations of command line instantiation that you may want to use, along with an explanation of what these combinations would do, and what you might use them for.

Although there are many possible option combinations of options, the most common are listed below:

`–ptnone –prelink –auto_include`	This is the default mode, which is suitable for most applications. On the first build of an application, the prelinker determines which source files should instantiate the necessary template entities. On subsequent rebuilds, the compiler automatically instantiates the template entities.
`–ptused`	This mode is suitable for small and medium sized applications. No prelinker pass is necessary. All referenced template entities are instantiated at compile time, and the linker removes duplicate functions. Dynamically initialized static data members are also handled correctly (by using a runtime guard to prevent duplicate initialization of such members).
`–ptused –prelink`	Use this combination when you have an archive or dynamic shared object (DSO) that has not been prelinked. When a DSO is built, it is automatically prelinked. When an archive is built, we recommend that you run the prelinker on the object files before archiving them. However, there are cases where a programmer may choose not to do so. For example, if an application is linked using multiple internal DSOs or archives, then you may choose not to prelink each DSO or archive, since that may create multiple instances of some template entities. When building an application using such archives or DSOs, you should use *–prelink* at compile time, even if the application is being built using *–ptused*. This is because the object files must contain not only instances of templates instances referenced in the compilation units, but also instances of template entities referenced in archives and DSOs.
`–ptall –no_prelink`	Use this combination when you are building a library of instantiated templates. For example, consider if you have a "stack" template class containing various member

functions. You may choose to provide instantiated versions of these functions for various common types (for example, int, float, and so on) and the easiest way of instantiating all member functions of a template is to use *−ptall*.

−ptnone −no_prelink

Use this combination if you are using template entities that are pre−instantiated.
For example, suppose you are using templates, but know that all of your referenced template entities have already been pre−instantiated in a library such as described in the previous example. In this case, you do not need any templates instantiated at compile time, and you should turn off automatic instantiation.

−auto_include

Use this option if you are using template implementation files that are not explicitly included.
Most source code written for *cfront* style compilers does not usually include template implementation files, because the *cfront* prelinker does this automatically. The *−auto_include* option is the default mode, because you want to compile *cfront* style code, but still instantiate templates at compile time (which implies finding template implementation files automatically).

−no_auto_include

Use this option if you are using template implementation files that are explicitly included.
Source code written for compilers such as Borland/C++ includes all necessary template implementation files. Such source code should be compiled with the *−no_auto_include* option.

−ptnone −no_prelink

Use this combination if all your template instantiation is done through the use of pragmas.
By using these options, you guarantee that nothing will be instantiated unless an explicit pragma is provided.

Pragmas for Template Instantiation

You can use pragmas to control the instantiation of individual or sets of template entities. There are three instantiation pragmas:

instantiate	Causes a specified entity to be instantiated.
do_not_instantiate	Suppresses the instantiation of a specified entity. Typically used to suppress the instantiation of an entity for which a specific definition is supplied.
can_instantiate	Allows (but does not force) a specified entity to be instantiated in the current compilation. You can use it in conjunction with automatic instantiation to indicate potential sites for instantiation if the template entity turns out to be required.

The arguments to the instantiation pragma may be

- a template class name, such as **`A<int>`**
- a member function name, such as **`A<int>::f`**
- a static data member name, such as **`A<int>::i`**
- a member function declaration, such as void **`A<int>::f(int, char)`**
- a template function declaration, such as **`char* f(int, float)`**

A pragma directive in which the argument is a template class name (for example, **`A<int>`**) is the same as repeating the pragma for each member function and static data member declared in the class.

When you instantiate an entire class, you may exclude a given member function or static data member using the *do_not_instantiate* pragma. See the example below:

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

You must present the template definition of a template entity in the compilation for an instantiation to occur. (You can also find the template entity with implicit inclusion.) If you request an instantiation by using the *instantiate* pragma and no template definition is available or a specific definition is provided, you will receive a link−time error. For example:

```
template <class T> void f1(T);
template <class T> void g1(T);
void f1(int) {}
void main()
{
    int i;
    double d;
    f1(i);
    f1(d);
    g1(i);
    g1(d);
}
#pragma instantiate void f1(int)
#pragma instantiate void g1(int)
```

f1(double) and *g1(double)* are not instantiated (because no bodies were supplied) but no errors are produced during the compilation. If no bodies are supplied at link time, you will receive a linker error.

You can use a member function name (for example, *A<int>::f*) as a pragma argument only if it refers to a single user−defined member function. (In other words, not an overloaded function.) Compiler−generated functions are not considered, so a name may refer to a user−defined constructor even if a compiler−generated copy constructor of the same name exists.

You can instantiate overloaded member functions by providing the complete member function declaration. See the example below:

```
#pragma instantiate char* A<int>::f(int, char*)
```

The argument to an instantiation pragma may not be a compiler−generated function, an inline function, or a pure virtual function.

Specialization

NCC supports *specialization*. In template instantiation, you specialize when you define a specific version of a function or static data member.

Because the compiler instantiates everything at compile time when the *−ptused* option is specified, a specialization is not seen until link time. The linker and runtime loader select the specialization over any non−specialized versions of the function or static data member.

See **"Pragmas for Template Instantiation"** for information on how to suppress the instantiation of a function. You may find this useful if you intend to provide a specialization in another object file and the non−specialized version cannot be instantiated.

Building Shared Libraries and Archives

When you build a shared library or archive, you should usually instantiate any template instances that could be needed.

The prelinker is automatically run when building a shared library, but it must be run manually when building an archive. Follow the steps below to build your archive:

1. Enter the command **`/usr/lib/DCC/edg_prelink a.o b.o`**

   This instantiates any templates needed by these object files.

2. Enter the command **`ar cr libtest.a a.o b.o`** to build the archive.

Limitations

There are some limitations on template instantiation in the Delta/C++ environment:

- A template specialization that exists in an archive may fail to be selected.

  If you define a specialization within an object file that exists in an archive, and that object file does not satisfy any references (other than the reference to the specialization), then the object file is not selected. Any function generated from a template that appears before the archive will be used, although a specialization should take precedence over a generated function.

The following conditions have to be present for the bug to occur:

- A template member needs to be specialized.
- The specialization must live in an archive element.
- A non−specialization of the template member must live in an object file seen by the linker. For a non−specialization to live in an object file, *−ptused* must have been specified (in other words, not the default mode).
- Nothing else that exists in the archive element is referenced; that is, the specialization is probably the only thing in the object file.

You can use either of the following two workarounds:

- Force the archive element to be loaded by defining some dummy global within it, and passing the *−u* option to the linker to force an undefined reference to the dummy global.
- Use a *.so* (that is, a dynamic shared object) instead of an archive. The runtime loader will correctly select specializations from dynamic shared objects.

- There is no link time mechanism to detect changes in template implementation files or to re−instantiate those template bodies that are out of date when you use the *−ptused* option.

  Since *Makefiles* usually makes object files dependent on the *.h* files where templates are defined, *make* may not enable you to rebuild the right set of object files if you modify a template implementation file. To make sure you rebuild all files that instantiate a given template when the template body changes, you must follow the steps below.

  1. Use the *−MDupdate* option at compile time to update a dependency file (usually called *Makedepend*). The compiler lists dependencies for all applicable *#include* files, including template implementation files that are implicitly included.
  2. Make sure that your *Makefile* includes this dependency file. See the *DCC(1)* and *make* reference pages for more information on how to include files within a *Makefile*.
- The only object files that the prelinker can recompile are object files that have not been renamed after they were originally compiled. In particular, the following limitations apply:
  - The prelinker cannot recompile any object file that exists in an archive, since putting an object file in an archive is equivalent to renaming it. It is recommended that you run the prelinker on object files before putting them in an archive. A similar restriction applies to dynamic shared objects (see **"Building Shared Libraries and Archives"**).
  - The prelinker cannot compile an object file if it was renamed after being compiled. For example, consider the following command line:

    ```
    yacc gram.y NCC −c y.tab.c mv y.tab.o object.o
    ```

    The prelinker does not know how to recompile *object.o*. If *object.o* contains unresolved template references that will not be satisfied by any other objects, you must use the *−ptused* option when compiling, or explicitly invoke the prelinker on the object file before moving it.

CC −64 Template Instantiation

The instantiation method for *CC −64* is somewhat different from that of *NCC*. *CC −64* currently does not support automatic template instantiation. The *NCC* options for template instantiation are not recognized by *CC −64*; instead, it behaves as if the options *−ptused −no_prelink −no_auto_include* had been selected. That is, the following is true:

- Any template entities used in a compilation unit are instantiated. This works much as the −ptused option, described in **"Command Line Options for Template Instantiation"** .
- Pragmas to control the instantiation are supported.
- There is no implicit inclusion of template definition files.
- If you plan on specializing a template function instantiation, you must set **`#pragma do_not_instantiate`** if it is likely that the compiler generated instantiation will contain syntax errors.
- The prelink mechanism is not supported.
- The associated *.ii* files are not created or used.

Template Language Support

The language support for templates in the Delta/C++ environment is more extensive than for *cfront*. Some of the additional template language constructs supported by Delta/C++ are listed below:

- You may use nested classes, typedefs, and enums in class templates, including variant typedefs and enums. (A variant member type depends on the template parameters in some way.)
- You may use floating point numbers, pointers to members, and more flexible specifications of constant addresses.
- You may use default arguments for class template non−type parameters. For example:

  ```
  template <int I = 1> class A {};
  ```

- You may allow a non−type template parameter to have another template parameter as its type. For example:

  ```
  template <class T, T t> class A {
  public:
   T     a;
   A(T init_val = t) { a = init_val; }
  };
  ```

- You may use what are essentially template classes instantiated with the template parameters of other class or function templates.

  ```
  template <class T, int I> struct A {
      static T b[I];
  };

  template <class T> void f(A<T,10> x) {}
  template <class T> void f(A<T, 3> x) {}

  void main()
  {
      A<int,10> m;
      A<int,3> n;
      int i = f(m);
      int j = f(n);
  }
  ```

  The function template would be considered tagged twice by *cfront*, and the code calls tagged ambiguous by the Borland/C++ compiler.

- You may use circular template references. For example:

  ```
  template <class T> class B;
  template <class T> class C;

  template <class T> class A { B<T> *b; };
  template <class T> class B { C<T> *c; };
  template <class T> class C { A<T> *a; };

  A<int> a;
  ```

  cfront generates an error on this code.

- *NCC* and *CC −64* are more consistent than other C++ compilers about where a class template name must be followed by template arguments. For example:

  ```
  template <class T> struct X {
      X();
      ~X();
      X*x;
      int X::* x2;
      void f();
  ```

```
    void g(){ X x;}
};

struct X<char> {
    X();
    ~X();      // Borland error
    X*x;      // Borland error
    int X::* x2;   // Borland error
    void f();
    void g(){ X x;}  // Borland error };

template <class T> void X<T>::f()
{
    X x;      // cfront error }

void X<char>::f()
{
    X x;      // cfront & Borland error
}

X<int> x;
X<char> xc;
```

cfront allows *X* to be used as a type name in the inline body of *g* but not in the out−of−line body of *f*. Borland/C++ uses one set of rules for class templates and a different set of rules for specializations. With *NCC* and *CC −64*, you may use *X* in all of the cases shown.

- You may use forward declarations of class specializations.
- You may use nested classes as type arguments of class templates.
- You may use default arguments for all types of function templates, including arguments based on template parameter types. For example:

```
template <class T> void f(T t, int i = 1) {}
template <class T> void f(T t, T i = 1) {}
```

Glossary

Terms shown in *italics* indicate glossary items, as well as buttons, file names, anconventions such

access

public, private, or protected members of a class.

base class

Parent of a class.

data

In C++, data member of a class. A variable that contains state information for a class. A field of a class that is not a method.

database

Compiler−generated static analysis database built from a fileset in the static analyzer.

delta compatible changes

Modifications to a class that may be made without requiring recompilation of any client code.

dynamic class

A class whose layout and location are determined at link time.

instantiate

In C++, declare an object of type *classname*. The result is an instance or an object.

internal dynamic class
A class that is dynamic because it inherits or contains a dynamic class.

mangle
Encoding a function name to support overloading.

member function
C++ term. See method.

members
Either or both data and methods belonging to a class (class members).

member extension
Adding member functions and variables to a class.

member promotion
Promoting a member to a non−virtual base class.

member reordering
Reordering members within a class.

method
C++ function that is a member of a class or structure data type. Also known as a member function.

object
C++ term. See instantiate.

override changing
Adding members that override those provided by base classes (whether or not the classes are virtual).

private
In C++, access to the class member is restricted to the class in which it is defined, friend classes, or friend functions.

protected
In C++, access to the class member is restricted to the class (and all derived classes) in which it is defined, friend classes, and friend functions.

public
In C++, access is open to any method or function.

specialization
In template instantiation, defining a specific version of a function or static data member.

templates
A description of a class or function that is a model for a family of related classes or functions.