C++ Programmer's Guide

Document Number 007-0704-070

CONTRIBUTORS

Written by Douglas B. O'Morain
Illustrated by Douglas B. O'Morain
Edited by Christina Cary
Production by Derrald Vogt
Engineering contributions by Rune Dahl, David Henke, Stuart Liroff, Michey Mehta, Roy Mittendorff, C. Murthy, Anil Pal, Andrew Palay, Krishna Sethuraman, Ravi Shankar, Shankar Unni, and John Wilkinson

C++ Programmer's Guide
Document Number 007-0704-070

Contents

List of Figures

List of Examples

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 involves *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, “DCC: the Delta / C++ Compiler,” describes *DCC*, the C++ compiler that allows you to use dynamic classes.
- Chapter 5, “Smart Build,” contains information on Smart Build, a feature that allows *DCC* to automatically determine the nature of changes in header files between compile runs, and to recompile only what actually needs to be recompiled.
- Chapter 6, “Code Examples,” provides code examples for *cfront* compatibility.
- Chapter 7, “Common Pitfalls,” discusses some common problems with C++ libraries and how to diagnose and solve them.

- Chapter 8, “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 Silicon Graphics implementation of the C++ language.

- *C++ Language System Overview* 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* contains a general description of the C++ language.
- *C++ Language System Library* 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 manual provides related information that you may need when using the Silicon Graphics C++ environment.

- *MIPSpro Compiling and Performance Tuning* discusses how to compile, and tune the performance of programs written in the Silicon Graphics development environment (C, Fortran, and C++).
- *dbx User’s Guide* discusses how to debug your code in the Silicon Graphics development environment.

Conventions Used in This Guide

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

- **Bold**—Functions, option flags, and classes
- *Italics*—Filenames, 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" on page 1 discusses the different Silicon Graphics C++ compilers for IRIX 6.0 and 5.3 systems.
- "Comparing CC to cfront" on page 3 compares the Silicon Graphics C++ to other C++ environments.
- "Using the Compilers" on page 4 discusses the differences between the 32- and 64-bit versions of the Silicon Graphics compilers, shows the command lines for the compilers, and gives some examples of typical command lines.
- "cfront Compatibility" on page 8 discusses the restrictions on C++ code that are enforced by the Silicon Graphics C++ environment, but were not enforced by *cfront*.
- "C++ Libraries" on page 10 discusses the C++ libraries in the Silicon Graphics C++ environment.
- "Debugging" on page 10 discusses the Silicon Graphics C++ debugging environment.

Silicon Graphics C++ Environment

The Silicon Graphics C++ environment is available in two varieties, targeted for IRIX 6.0 and 5.3 systems. The 6.0 32-bit compiler and the 5.3 compiler are functionally identical. See Figure 1-1 for details.

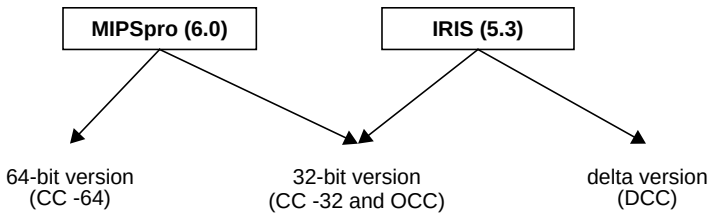

Figure 1-1 Silicon Graphics C++ Environment

Note: *CC -64*, the 64-bit version of the *CC* compiler, can only be run on 6.0-based systems.

MIPSpro 6.0 Compilers

As shown in Figure 1-1, there are 32- and a 64-bit versions 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 available. The supported C++ compilers for the 6.0 system are listed below:

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

On 64-bit hardware, *CC* generates 64-bit code by default (without using the *-64* extension explicitly), while on 32-bit hardware, it generates 32-bit code by default.

IRIS 5.3 Compilers

There are also two versions of the C++ compiler for the 5.3 operating system: *CC* 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 5.3 system are listed below:

CC	32 bit native C++ compiler. Functionally identical to 6.0 32-bit *CC*.
DCC	Delta / C++ compiler. The delta feature enables you to use dynamic classes to minimize the need for recompilation upon changing a class, thereby greatly increasing productivity. For complete information on *DCC*, see Chapter 4, "DCC: the Delta / C++ Compiler."
	Note: Most *CC* information in this guide applies to *DCC* as well. The differences from *CC* are described in Chapter 4.
OCC	32-bit C++ compiler, based on C++ to C translation using *cfront*.

For complete details on *cfront* compatibility, see "cfront Compatibility" on page 8. For further details on the commands themselves, refer to the *CC(1)*, *OCC(1)*, and *DCC(1)* reference pages.

Comparing CC to cfront

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

Native Compiling

CC is a native compiler that is a drop-in replacement 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). *CC* 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 *CC* 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.

CC instantiates 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

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

Note: Exception handling is not supported in *CC*. *CC* 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* and *OCC* command lines (and some of the more commonly used options), and contains some examples.

The default compiler depends on your hardware: on 64-bit systems, *CC* defaults to *-64* mode; on 32-bit systems, *CC* defaults to *-32* mode. If you use *CC* with options supported by *OCC* but not supported by standard *CC*, or you use *CC* with the *-use_cfront* option, you invoke *OCC*.

64- Versus 32-Bit Compilation

The 64- and 32-bit versions of *CC* 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 *CC -32* 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. 64-bit objects can only be created on 6.0-based systems. You can do this as follows:

- Specify the *-64* option on the IRIX 6.0 command line to compile source files for 64-bit objects. With *-mips4*, this is the default for the MIPSpro compilers installed on an IRIX 6.x system
- Specify the *-32* option on the IRIX 6.0 command line to compile source files for 32-bit objects. With -mips1, this is the default for the MIPSpro compilers installed on an IRIX 5.x system.

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

Some additional differences between the 64- and 32-bit version of *CC* are listed below.

- *CC -64* and *CC -32* support different template instantiation options.
- The warning options used by the -woff option are different in *CC -64* and *CC -32*.

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.

CC Command Line

The command line for *CC* is shown below.

```
CC [ option ] . . . file . . .
```

CC compiles with many of the same options as cc(1). *CC -64* is the default on 6.x (64-bit) systems, and *CC -32* is the default on 5.x (32-bit) systems.

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

See the *CC(1)* reference page for more information.

DCC Command Line

The command line for *DCC* is

```
DCC [ option ] ... file ...
```

Note: DCC is supported only in IRIS 5.3 systems, and in 32-bit mode only.

DCC compiles with many of the same options as cc and *CC*. See the *DCC(1)* reference page for supported and unsupported *CC* options.

By default, *DCC* does not enable dynamic classes. You can make your classes dynamic

- selectively or for all classes defined in the current directory
- selectively or for all classes defined in the current hierarchy
- selectively or for all classes defined in the source code

For complete details on dynamic classes, see “Dynamic Classes” on page 26.

OCC Command Line

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

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

You may also use the following command line:

```
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 -use_cfront program1 program2 . . .
  ```

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

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

- To compile with delta capability and the Smart Build facility, the command line is

  ```
  DCC -smart program1 program2 . . .
  ```

  (For information on Smart Build, see “Smart Build,” in Chapter 4.)

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 *CC*.

For examples of code that were valid under the *cfront* environment but are invalid in the Silicon Graphics C++ environment, see “cfront Compatibility Examples” on page 53. Specific examples are cited in the following list of restrictions.

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:

- 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 6-1.
- You must have an explicit declaration of a constructor or destructor in the class if there is an explicit definition of it outside the class. See Example 6-2.
- You may not delete a pointer to a const. See Example 6-3.
- You may not pass a pointer to volatile data to a function that is expecting a pointer to non-volatile data. See Example 6-4.
- 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 6-5.

- You may not use redundant type specifiers. See Example 6-6.
- 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 6-7.
- You may not assign a 0 to a pointer, if the 0 is the right-expression of a comma operator. See Example 6-8.
- You must not use the same identifier for more than one formal argument in a function definition.
- The Silicon Graphics compiler will *mangle* member functions declared as extern "C" differently from *cfront*. *CC* 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".

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 *dbx User's Guide*.

Chapter 2

Compiling, Linking, and Running C++ Programs

This chapter contains the following major sections:

- "Compiling and Linking" on page 11 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" on page 15 describes the command line options that can be provided to the C++ translator.
- "Object File Tools" on page 17 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" on page 15 in this chapter.
filename.C	is the name of the file that contains the C++ source statements. The filename must end with one of the following acceptable suffixes: *.C, .c++, .cc, .cpp, .CPP, .cxx* 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
```

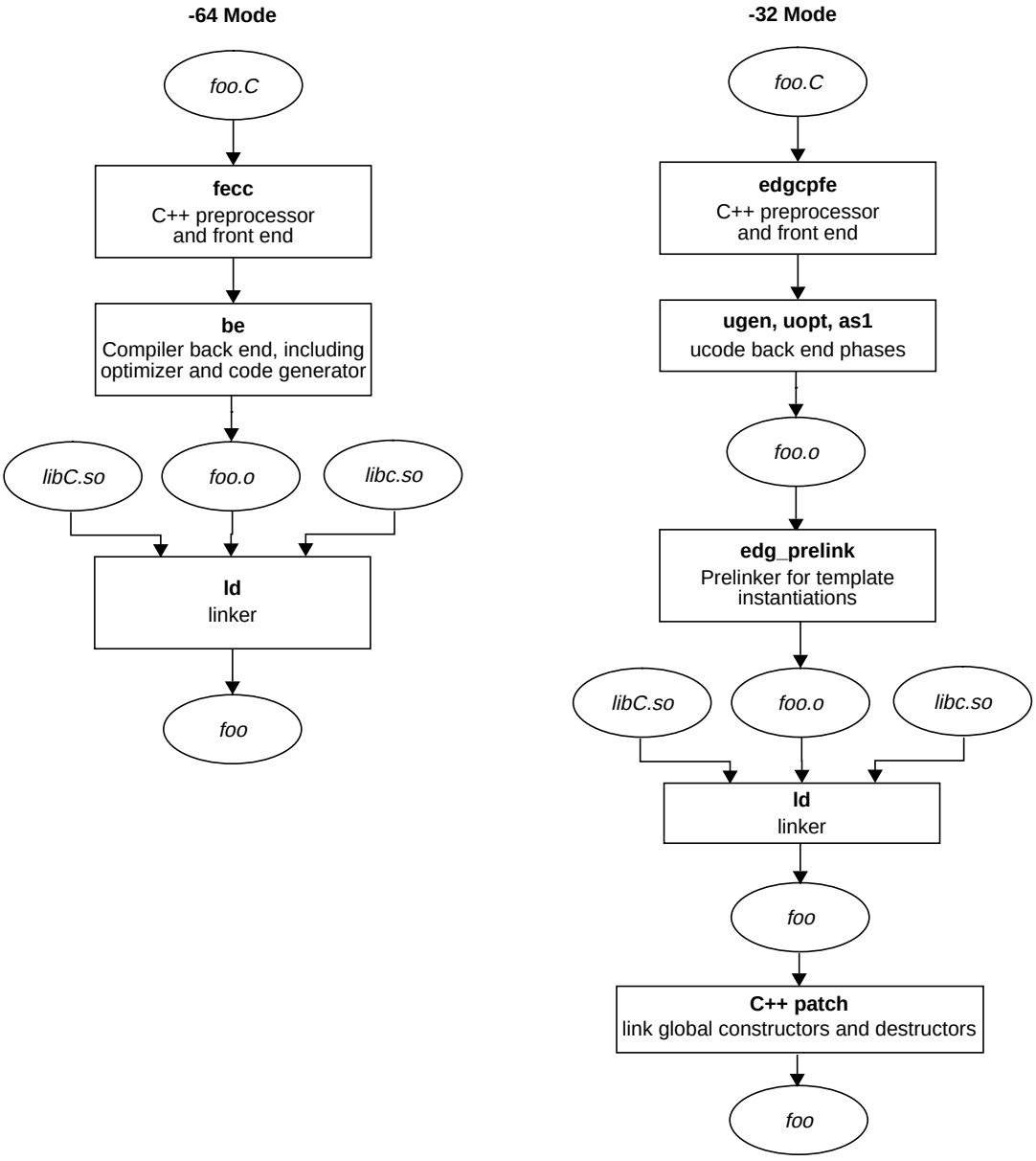

Figure 2-1 The Compilation Process

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++*, *.c*, *.cc*, *.cpp*, *.CPP*, *.cxx* or *.CXX*.
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 function **main()** is called, 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” on page 10.

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), *-show0* (*-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)
Warn about all questionable constructs. Without the *+w* option, the translator issues warnings only about constructs that are almost certainly problems.

+p (*-32* 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, by default anachronisms are disallowed and the stricter definition is the default enforced.

-use_cfront (*-32* mode)
Use *OCC* instead of *CC*.

-cfront (*-64* mode)
Compile in *cfront* compatibility mode. This is the default in *-32* mode. The *+pp* option will disable *cfront* compatibility mode.

–nocpp	Skip the preprocessing stage.

Object File Tools

For information on the object file tools available to you, consult the *MIPS Compiling and Performance Tuning 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 output 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*.
elfdump	The *elfdump* tool lists the contents (including the symbol table and header information) of an ELF-format object file. See the *elfdump(1)* reference page for more information.
stdump	The *stdump* tool outputs a file of intermediate-code symbolic information to standard out. See the *stdump(1)* reference page for more information.

Chapter 3

Interfaces

This chapter contains the following major sections:

- "Using Other Language Libraries from C++ Programs" on page 19 describes how to link the libraries from other languages into your C++ program.
- "C Function Declarations in C++ Programs" on page 20 describes how to use declarations in the Silicon Graphics C++ environment.
- "Using C++ Libraries From C Programs" on page 21 describes 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 prototypes 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.so*, 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 **__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* or *ld*. Linking using *CC* ensures that the linked executable will have its C++ global objects correctly initialized on program start-up.

If your program (or DSO) includes C++ object files that have global objects that need to be constructed at program start-up, link your program (or DSO) using *CC*. (For information on DSOs, see “Dynamic Shared Objects” in the *MIPS Compiling and Performance Tuning Guide*.)

Chapter 4

DCC: the Delta / C++ Compiler

This chapter describes *DCC*, the Delta / C++ compiler that allows you to use dynamic classes. It contains the following sections:

- "DCC Enhancements"
- "Dynamic Classes"
- "Class Library Modification Compatibility"
- "Running the Delta / C++ Compiler Tutorial"
- "Porting Your Code to Delta / C++"
- "DCC Limitations"

DCC Enhancements

DCC, the Delta / C++ compiler, is a native compiler that contains a number of enhancements over the standard Silicon Graphics C++ compilation environment. *DCC* allows you the option of using dynamic classes and modifying class libraries without client recompilation. *DCC* also contains a few enhancements over *CC*, but the primary difference is the use of *dynamic classes*. A dynamic class is a class whose layout and location are determined at link time.

DCC and *CC* use the same compiler front end, and enforce many of the same restrictions to approved C++ coding style that *cfront* overlooks. *CC* uses a more conventional code generation style than *DCC*. *DCC* predefines the *_DELTA* symbol during the preprocessor phase of compilation. *DCC* creates the file *delta_preload.h*, which contains the *_DELTA* declaration.

Note: *DCC*- and *CC*-produced object files are compatible.

Resolving Object References at Link Time

Current C++ systems have been designed without considering the needs of environments that make use of shared libraries or dynamic loading. If you release a new compatible version of a library or dynamically loaded component, you must recompile the portions of the system that make use of the classes defined in the new component. This is because current C++ systems resolve all object references at compile time.

The Delta / C++ compiler is a runtime / linktime extension to C++ that allows you to make compatible changes to class definitions with minimal recompilation, minimal restrictions on the use of the language, and minimal runtime overhead. Client applications linked against a shared library will continue to run, without recompilation, even when a new, compatible version of the shared library is released. Delta / C++ resolves all object references at link time.

Program Development

DCC allows you to make changes to a system under active development without having to recompile the entire system in order to test the change or to continue development.

For example, in large systems, the cost of changing a class definition is often prohibitive, not only to the developer making the change, but also to developers using the change. You may have to rebuild the entire system. Developers will frequently work around a problem instead of solving it. In the end, few improvements are made to the base system.

The *DCC* extensions make it possible to recompile only a small set of files and resume work.

Smart Build

Smart Build extends the Delta / C++ compiler to reduce build times for C++ sources significantly. Invoking the *-smart* flag works seamlessly with any existing build environment (for example, *make*). Smart Build contains two interdependent mechanisms: precompiled headers (available in *DCC* and *CC*), and Smart Build itself, which uses these headers to calculate the

differences between two versions of a header file. Smart Build keeps *make* from recompiling all files that include a modified class definition.

For further information on Smart Build, see Chapter 5, “Smart Build.”

Support for New Versions of C++ Shared Libraries

DCC allows applications linked against a shared library to continue to work when a new version of the library is installed. For example, consider a typical scenario:

1. Silicon Graphics, Inc. ships several C++ shared libraries (DSOs).
2. A developer builds a client application linked against those libraries.
3. A consumer purchases the client application.
4. Silicon Graphics, Inc. ships new versions of the shared libraries.

To allow the new shared libraries to continue to work with existing applications, developers often ship several versions of the shared libraries, each bearing the version as part of their name (for example, *lib/Inventor.so.1.1*, *lib/Inventor.so.1.2*). This practice has several disadvantages:

1. More disk space is required.
2. Less actual sharing occurs, because old client applications use the old version of the shared library, and new client applications use the new version.
3. Old client applications do not benefit from any improvements to the new shared library (for example, performance enhancements or bug fixes).
4. The producer of the shared library (for example, Silicon Graphics) has to ensure that all versions of the shared library are shipped. Consumers (users of the shared library) must also install all of the versions, since there is no way for them to know when they will need a particular library, and they would have to go back to their initial installation and find the right library if they didn’t install all of the versions.

With *DCC*, new versions of C++ libraries can be shipped with the assurance that they will work with existing client applications that were formerly bound with older versions of the C++ libraries.

In addition, most library providers have to expose a private implementation of your classes, because private members must be visible to the compiler in order to lay out the class. *DCC* allows you to remove that portion of the class definition and still have the code work correctly.

Dynamic Classes

DCC supports the use of dynamic classes. A dynamic class is one whose layout and location are determined at link time. This gives you the ability to make changes to libraries and applications without having to recompile your entire code hierarchy. You may add members, base classes, promote members, change the overrides, and reorder the members of a dynamic class. (For a complete description of what is allowed under the Delta / C++ environment, see “Delta-Compatible Changes” on page 34.)

Note: *DCC*, by default, considers a class to be non-dynamic.

Using Dynamic Classes

As a developer, you use classes in three situations (see Figure 4-1 for a graphical representation):

- classes that are imported from another library
- classes that are used internally only
- classes that are exported to a third party

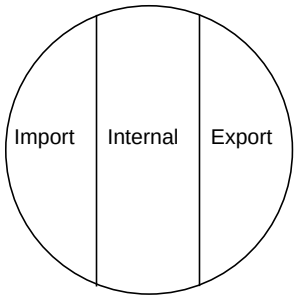

Figure 4-1 Using Classes

When would it be best for you to use dynamic classes?

Import classes	You would prefer these classes to be **dynamic** classes to help you avoid having to do massive recompilations when you must make changes to your code, but it is up to your library provider.
Internal classes	Use **non-dynamic** classes, unless you are doing code development using *-smart* (see Chapter 5, “Smart Build”).
Export classes	Use **dynamic** classes if you are planning to change them in future releases, and want to avoid affecting users.

The amount of classes that you import, use internally, and export varies depending on what type of code you are developing. For example, perhaps you are a library developer; you would be importing a few classes, exporting large numbers of classes, and using very few internal-only classes. See Figure 4-2 for an illustration of this distribution.

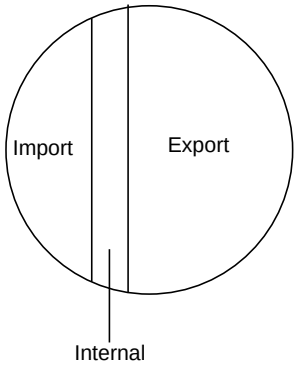

Figure 4-2 Library Class Use

Perhaps you are developing applications. In that case, you would be importing some classes, exporting none, but using many internally, as illustrated in Figure 4-3.

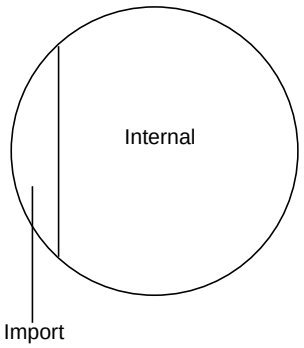

Figure 4-3 Application Class Use

Setting Classes to Be Dynamic

DCC makes classes dynamic in any of the ways listed below.

- You may make a class dynamic on a class-by-class basis with the **pragma** *dynamic_class*.
- You may use the **pragma** *this_directory_tree_is_dynamic* to enable dynamic classes for a given directory hierarchy. Any class that contains a non-inline member function in that hierarchy will be dynamic (unless you specifically make that class non-dynamic; see “Disabling Dynamic Classes” on page 31).
- Any class that derives from a dynamic class is made dynamic.
- Any class that contains one or more instances of dynamic classes is made dynamic.

Using the delta_preload.h File

When you execute *DCC*, the compiler first looks for a file named *delta_preload.h* in the directory that contains the file you are trying to compile. If *DCC* does not find *delta_preload.h*, it searches the parent directory. *DCC* continues to search for a *delta_preload.h* file until one is found, or until it reaches the root directory.

delta_preload.h ordinarily contains one of the following **pragmas**:

this_directory_tree_is_dynamic
: instructs *DCC* to make any class that contains a non-inline member function dynamic. The definition is located in a file in the directory hierarchy headed by the directory in which *delta_preload.h* is found.

this_directory_tree_is_not_dynamic
: instructs *DCC* to make classes non-dynamic. The definition is located in a file in the directory hierarchy headed by the directory in which *delta_preload.h* is found.

Searching up the directory hierarchy for *delta_preload.h* makes it easier for you to control when classes are dynamic for an entire project.

For example, in the directory hierarchy in Figure 4-4, you are compiling */usr/app/src/appfile.c*. If the only *delta_preload.h* file is found in */usr/app/src*, and it contains **`#pragma this_directory_tree_is_dynamic`**, then any class definition that contains a non-inline member function and is located in a file in */usr/app/src* is marked as being dynamic.

A class definition that contains a non-inline member function and is located in a file in */usr/app/include/incfile.h* is not considered dynamic. If the *delta_preload.h* file was moved to */usr/app*, classes containing a non-inline member function defined in files from the include directory are also dynamic.

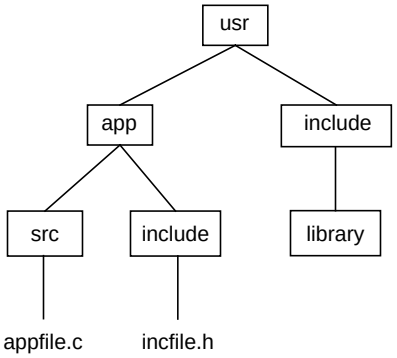

Figure 4-4 Directory Hierarchy for Dynamic Classes

The compiler searches for a *delta_preload.h* file for the source file being compiled and any file that is included during the compilation. This makes it possible for a library provider to make classes exported by that library dynamic without affecting classes that come from the application or other libraries.

You can do this by putting a *delta_preload.h* file that contains the *this_directory_tree_is_dynamic* **pragma** in the top of the directory tree that contains the include files for the library. For example, in Figure 4-4, putting a *delta_preload.h* file in */usr/include/library* would make any class dynamic, if that class has a non-inline member function defined in a file in that directory or any of its sub-directories.

When you are writing production code, you want classes to be non-dynamic by default. During the development phase, you probably want the classes to be dynamic. Using dynamic classes in conjunction with the smart build feature of the compiler greatly decreases the turn-around time when you make a change to a class definition. The best way to do this is to create a *delta_preload.h* file that looks like that in Example 4-1.

Example 4-1 Setting Dynamic Classes During Code Development

```
#ifdef CODE_DEVELOPMENT
#pragma this_directory_tree_is_dynamic
#else
```

```
#pragma this_directory_tree_is_not_dynamic
#endif
```

You must put this file at the root of the application code source directory (*/usr/app* in the directory hierarchy in Figure 4-4). Use the following options during code development:

```
-DCODE_DEVELOPMENT -g -smart,/usr/app/HDRS
```

Do not use the first two options during a production build.

Enabling Dynamic Classes in a Hierarchy

You can specify all the classes defined in an entire hierarchy to be dynamic by setting a **#pragma** in that directory's *delta_preload.h* file, as shown in the command below:

```
#pragma this_directory_tree_is_dynamic
```

The classes containing non-inline member functions defined in that directory are dynamic.

Enabling Dynamic Classes in a Source File

You can specify a class to be dynamic with a **pragma**:

```
#pragma dynamic_class
```

You can define dynamic classes by lists of one or more in either the source code itself or a header file.

Disabling Dynamic Classes

You can specify a class to be non-dynamic with a **#pragma** in several ways:

- You may define the **#pragma**

  ```
  #pragma nondynamic_class <class>
  ```

  where the **#pragma** appears before the class definition.

- You may define the **#pragma**

  ```
  #pragma nondynamic_class “regular expression”
  ```

where the only classes affected are those that appear after the pragma, and match those listed in the regular expression. (For more information on regular expressions, see the *regexp(5)* reference page.)

- You may define the **#pragma**

  ```
  #pragma nondynamic_class
  ```

  without a name if it appears within a class definition.

A class that is non-dynamic may have to be implemented as a dynamic class for various reasons (for example, the class contains a member or base class that is dynamic). Such classes are called internal dynamic classes, and may not be modified in future releases (that is, from a programmer's perspective, they are non-dynamic classes).

Note: You cannot apply the **nondynamic_class** pragma on a class that must be internally dynamic.

The preprocessor automatically defines the *__DELTA* symbol when using *DCC*; this allows you to conditionally compile Delta / C++-specific code changes (such as the addition of pragmas).

Dynamic Class Error Messages

The following are programming errors you may encounter when you use dynamic classes via the *DCC* extension.

- *DCC* generates an internal description of a class layout. If this description is not available when linking, you will see a linker error:

  ```
  Delta Error: Could not find the class definition for
  <class>. You possibly missed providing a definition for
  the first virtual method or the first non-inlined method.
  ```

  The class layout description is written to the appropriate object file. (The file is the one that corresponds to the source file containing the member function described in the error message.)

  The usual source of this error is that you declared a constructor but never provided a definition for it.

- When the DSO is linked into a program, the class description must be available, or you will see a warning:

```
Delta Warning: Could not find the class definition for
<class>
```

(For further information on DSOs, please see “Dynamic Shared Objects” in the *IRIX System Programming Guide*.)

- When your code is expecting a member function to be defined, but the definition was not found in the class definition, this message appears:

  ```
  Delta Error: Missing the member function <function> in
  <class>
  ```

  There are two common reasons for this problem:

  – You compiled with mismatched header files; that is, one source included an older version of the class definition lacking a member function added to the newer version of the same header.

  – The class definition itself was missing, in which case either of the first two messages appears.

  To fix the problem, compile with the correct header files.

- If a class has base classes, then the class description will refer to them. If, for some reason, one of the base class definitions could not be calculated, then you will see an error:

  ```
  Delta Error: invalid class <class>: derives from invalid
  base classes
  ```

- If two classes have the same name, and they are linked into an application or into a DSO, and the classes have a member function with the same signature, the linker will issue a warning that the member function is multiply defined. (The compiler cannot detect this condition, only the linker can.)

  If you make one or more of the classes dynamic when compiling with *DCC*, then the likelihood of a member function colliding increases, because dynamic classes always generate out-of-line constructors, destructors, and assignment operators. When this happens, the linker, when run under *DCC*, generates a warning:

  ```
  Delta Warning: Multiple definitions of <class> - ignoring
  new definition
  ```

Local Classes

Local classes are classes declared within functions. These classes may be internally dynamic only. There is never any need for a local class to be dynamic, since it cannot be accessed outside of the current compilation unit.

Nested Classes

Nested classes behave the same way as non-nested classes.

Class Library Modification Compatibility

DCC allows you to make various changes to a class library that are backwards-compatible with applications compiled with older versions of the library. You may make only these changes to dynamic classes.

Delta-Compatible Changes

The following changes to a class library are backwards-compatible with applications compiled with older versions of the library. These are the *delta-compatible changes.*

Member Extension
: You may add member functions and variables to a class without having to recompile any code that uses the class. This is true for public, protected, and private members.

Class Extension
: Base classes may be added to a class. If you have virtual or multiple inheritance, any class extension is possible. If you do not have virtual or multiple inheritance, class extension is limited in the following way:

- If the class is a root class, you may add only one base class.
- If there are no virtual base classes, and you have single inheritance, you can split a class into two classes.

In either of these two cases, you can allow for future multiple inheritance by setting a pragma:

```
#pragma allow_for_future_multiple_inheritance
```

Member Promotion

You may promote a member to a non-virtual base class. You can move functionality from a derived class to a non-virtual base class, so long as its type remains identical, and is not an inline function.

Override Changing

You may add members that override those provided by base classes (whether or not the classes are virtual).

Member Reordering

You may reorder members within a class. This has no effect on the interface being provided by the class.

See “Delta-Compatible Changes Examples” on page 59 for examples.

Delta-Incompatible Changes

The following changes to a class library are not backwards-compatible with applications compiled with older versions of the library. If any of the following restrictions apply to your code, you must recompile. These are the delta-incompatible changes.

- You cannot change the declaration of a member in any way (for example, changing a type from *short* to *long*). This includes the types of data members, function declarations, inline function bodies, typedef declarations, and enumeration declarations. The full set of restrictions is listed below.
  - You may not change the values of enumeration constants. However, you may add values to an existing enumeration provided you do not modify the values of any of the existing enumeration constants. See “Changing the Values of Enumeration Constants” on page 65 for an example.

  - You may not add or remove parameters from a member function, and you cannot change the return type of a function. You may add a default value to an existing parameter that did not already have one. See “Adding or Removing Member Function Parameters” on page 66 for an example.
  - You may not change the default parameters of a member. See “Changing Member Function Default Parameters” on page 66 for an example.
- You may add an overloaded function to a class only if the function you are adding cannot possibly be called by existing code. See “Adding Overloaded Functions to a Class” on page 67 for an example.
- You may not override an inline function defined in a base class with a new version of the function in a derived class. See “Base Class / Derived Class” on page 68 for an example.
- You may not override a global object in a base class when any derived classes reference that object. You also may not override functions with default arguments. See “Base Class / Global Object” on page 69 for an example.
- You may not change the class inheritance from single to multiple inheritance when adding a base class. See “Changing From Single to Multiple Inheritance” on page 70 for an example.
- You may not move members to an enclosing class. Delta / C++ allows you to promote a member to a base class, or override a member from a base class. However, you may not move a member to an enclosing class, or hide a member from an enclosing class. See “Moving Members to an Enclosing Class” on page 71 for an example.
- There are various restrictions on virtual functions, virtual base classes, and static members:
  - Virtual base classes may not be made non-virtual (and vice versa).
  - Virtual functions may not be made non-virtual (and vice versa).
  - Non-virtual functions may not be overridden by a virtual function.
  - Members may be promoted only to a non-virtual base class.

  - Static members may not be made non-static (and vice versa).
  - Static members may not be overridden by non-static members (and vice versa).
- You cannot make a class abstract by adding a pure virtual function.
- You may not move inline member functions from one class to another.
- If you define a class as a leftmost base class in the first release, it must continue to be the leftmost base class in all future releases.

See “Delta-Incompatible Changes Examples” on page 65 for examples.

Running the Delta/C++ Compiler Tutorial

The Delta / C++ product comes with a tutorial located in the directory */usr/demos/DeltaCC*. To run this tutorial, you must follow these steps:

1. Enter the command **`cd /usr/demos/DeltaCC`** to move to the demos directory.
2. Enter the command **`make delta`** to run a demonstration of Delta / C++.

This run tries to recompile only those files that we know are in need of recompilation, and shows how the test program picks up these changes without necessarily having to recompile the whole world.

Porting Your Code to Delta/C++

To port your *cfront*-compatible source code to the Delta / C++ environment, you must follow these steps:

1. Change your makefiles to invoke *DCC* with options it will recognize.
2. Decide which of your classes will be dynamic. (You can define classes as dynamic or not dynamic with the *delta-preload.h* file and pragmas. See “Setting Classes to Be Dynamic” on page 28 for more information.)
3. Correct source code that is not accepted by *DCC*.

Changing Your Makefiles

To change your makefile you must

- Use *DCC* instead of *CC* (the *cfront* driver) when compiling.
- Use *DCC* to link an executable or DSO; invoking the linker (*ld*) directly on C++ object files is not recommended.
- Remove any options (such as *-float*) that are no longer supported. See the *DCC(1)* reference page for a list and description.
- Use the *+w* or *-fullwarn* option to get warnings about questionable constructs.
- Enable SmartBuild with the *-smart* option. You may want to place all precompiled header files into the same repository directory, so put **`-smart`**, **`ILDUMPS`** in your Makefile. (*ILDUMPS* is a directory of your choice.)
- Invoke the prelinker before running the *ar* command if you are building an archive (a *.a* file) out of object files that use templates. See “Building Shared Libraries and Archives” on page 89 for more information.

Correcting Your Source Files

When you move from *cfront* to DCC, there are several areas that may cause you problems. These areas are *cfront* incompatibilities, *DCC* limitations, and added *DCC* warnings.

cfront Incompatibilities

By default, *DCC* tries to accept everything that *CC* did, even if these are sometimes illegal (and *cfront* simply doesn’t catch it). There are some things, however, that *DCC* does not allow, and instead issues an error. For details, see “Compatibility Restrictions” on page 8.

The functionality of the *+p* option with *DCC* is identical to that with *CC*. However, you may want to use the *+pp* option instead. The *+pp* option disables *cfront* compatibility as well as disabling anachronisms. The *+pp* option enforces a strict interpretation of the current C++ language

specification. The errors and warnings generated, when addressed and corrected, will give you more correct and reliable software.

DCC Limitations

Due to implementation constraints, *DCC* disallows certain constructs. See “DCC Limitations” on page 39 for details.

Added DCC Warnings

DCC issues more warnings than *cfront*. While these should be addressed and corrected if possible, you may want to use the *-woff* option for those warnings you’re convinced are safe to ignore.

DCC Limitations

DCC has a few limitations when the **_DELTA** facility is enabled:

- Since the size of a dynamic class is no longer known at compile time, calling **sizeof** for a dynamic class no longer yields a compile time constant. In the following example, **`sizeof(myclass)`** cannot be used as a constant to set the buffer size for **`char buf`**:

  ```
  class myclass
  {
    public: void A();
    int a;
  };

  void myclass::A() {}

  char buf[ sizeof(myclass)]; // \Gets error.
  ```

  Taking the size of a dynamic class in an expression that is evaluated at run time will compile and work.

- The size of a dynamic class cannot exceed 32767 bytes.
- Constructors, destructors, and assignment operators for dynamic classes are always out of line.
- A union may not have a member that has a constructor. Since *DCC* always ensures that all dynamic classes have a constructor, objects of

such classes may not be a member of a union. Suppose you have the following code:

```
class dynamic {
public:
  foo();
};

union xyz {
  int x;
  int y;
  dynamic member; // Error 375.
  xyz();
};
```

cfront accepts this code, but *DCC* generates the following error:

```
error(375): invalid union member -- class “dynamic” has a
disallowed member function
dynamic member; // Error 375.
        ^
```

- You cannot pass a dynamic class to a “`...`” parameter. When the formal parameter for a function is “`...`”, the corresponding actual parameter in a function call cannot be a dynamic object. A class object is passed to a “`...`” parameter “by value” (that is, no copy constructor is ever run), but since we do not know the size of a dynamic object, it cannot be passed by value on the stack.

  Suppose you have the following code::

```
struct dynamic {
#pragma dynamic_class
  dynamic();
};

dynamic object;

void f(int,...);

void main(){
  f(5,object);
}
```

Compiling this code with *DCC* generates the error below.

```
“bug.c”, line 11: error(671): dynamic class cannot be
passed to a ... parameter f(5,object);
                            ^
```

- *_builtin_alignof* is not a compile time constant. When you apply *_builtin_alignof* to a dynamic class, the alignment of the class may subsequently be changed by the addition of new data members. It is illegal to use *_builtin_alignof* in contexts where a compile time constant is needed.

  For example, you cannot have the declaration of an array as in the code sample below.

  ```
  int x[__builtin_alignof(dynamic_class)];”
  ```

  This restriction is similar to the restriction imposed on “sizeof”.

Chapter 5

Smart Build

This chapter describes Smart Build, a feature that allows the Delta/C++ compiler *DCC* to automatically determine the nature of changes in header files between compile runs, and to recompile only what actually needs to be recompiled, according to the Delta-compatibility rules.

Smart Build contains two interdependent mechanisms: precompiled headers (available in *DCC* and *CC*), and Smart Build itself, which uses these headers to calculate the differences between two versions of a header file.

This chapter contains the following sections:

Understanding Smart Build

The Smart Build facility in the Delta/C++ compiler allows the compiler to automatically determine the nature of changes to a header file, and recompile only a compilation unit (that is, generate object code for it) if necessary. (See the definition of Delta-compatible changes in the previous chapters.)

Smart Build allows you to continue using your current build mechanisms of choice (like *make(1)*), and still take advantage of Delta/C++'s benefits of reduced recompilation. For example, *make*, on noticing that a header file has changed, attempts to launch a recompile of every compilation unit that includes that header file. Smart Build allows *make* to examine the exact nature of the change, and generate new object code only when it is necessary.

When the change is a Delta-compatible one, this allows the compiler to terminate the compile for most compilation units at the early stages (typically 1-2 seconds) and merely touch the object file, thus dramatically speeding up the recompile.

To do this, the compiler builds binary precompiled files for each of the header files used in the compilation—one for each source file. When a file is modified, the compiler rebuilds the binary precompiled file, and can compare it for differences against the previous version of the binary file.

Smart Build has two interdependent mechanisms: precompiled headers, which are available both in *DCC* and *NCC*, and the Smart Build facility, which uses these precompiled headers to calculate the differences between two versions of a header file.

Invoking the Smart Build Facility

Smart Build can be used with both *NCC* and *DCC*. Smart Build skips rebuilding more object files in *DCC*, but using it with *NCC* still gives you a performance increase.

Smart Build and DCC

Invoke the Smart Build facility by adding the *-smart* option to the *DCC* command line:

```
DCC -smart[,repository_dir] [options] . . . file . . .
```

repository_dir is the directory where the compiler keeps the precompiled binary versions of the header files. The default is *./ILDUMPS*. If the specified directory does not exist, the compiler will attempt to create it with *mkdir(2)*.

When a header file is touched in any way and a rebuild is performed—usually using *make(1)*—the first invocation of the compiler (for the first compilation unit recompiled by *make(1)*) performs the following functions:

- notices the modified header file,
- rebuilds the binary precompiled file

- computes the differences between this version of the binary file and the previous version.

If any changes are not Delta-compatible, the compiler will generate new object code for the compilation unit, otherwise it prints a message:

```
Remark(2): Smart Build: recompilation skipped
```

The compiler then touches the object file (so that make will not attempt to repeatedly recompile the file in future invocations).

If you invoke the compiler in the same run of *make(1)* for the remaining compilation units that also include this modified file, the compiler will

- see the new version of the binary precompiled file
- notice that the file has been rebuilt since the last time the object file for this compilation unit was created
- examine the recorded changes from the previous version of the header file
- see if any of the changes affect this compilation unit

If not, the compiler quickly prints the above message and touches the object file, so that the compiler will generate new object file for this compilation unit, too.

Smart Build and NCC

The greatest benefits of Smart Build are available with *DCC*, where the compiler has a greater leeway to skip rebuilding object files if changes are deemed to be Delta-compatible. However, it also benefits *NCC* to a somewhat lesser extent.

When using Smart Build with *NCC*, most of the rules for compatible changes do not apply. For instance, adding or reordering members in a class causes any compilation unit using that class to get its object file rebuilt.

There is a small set of changes that are considered compatible:

- You may reformat or add comments without changing the order of the declarations in a file.

- You may add an extern variable or function declaration.
- You may add a type declaration.

If the compiler determines that the changes meet these criteria, it will skip recompiling the compilation units that include that object file.

Precompiled Header File Mechanism

The precompiled header file mechanism used in the Smart Build facility has some major advantages:

- It is fully automatic.
- The pre-compilation is done on a per-header file basis, so the users need not modify their source in order to use this mechanism.

The mechanism is extremely sensitive to external factors like macro definitions, especially if certain macros have different settings for different compilation units, including the same header file.

There are circumstances in which the compiler has to rebuild a header file (or use the source header file) even if it has not changed:

- You use a macro in the header file with a different setting than when it was last precompiled.
- The compiler detects a condition due to which it cannot build a precompiled header file (usually an implementation compromise — see “Conditions for Not Building Precompiled Headers” on page 48).
- There is an explicit directive in the header file that tells the compiler not to precompile the header file.

Even if you have to rebuild object files (either if the changes are not compatible, or object files are missing), the precompiled header mechanism still provides a significant performance increase.

If you don’t have to rebuild precompiled headers for any other reasons (see above), then you can expect to see about a 20% improvement in compilation time.

Causes of Inefficiencies in the Precompiled Header Mechanism

Most of these inefficiencies are due to macro dependencies. For example, consider the code segment in Example 5-1.

Example 5-1 Inefficiencies Due to Macro Dependencies

```
#ifndef _SIZE_T_DEFINED
#define _SIZE_T_DEFINED 1

typedef unsigned int size_t;

#endif
```

Code such as this frequently appears in many of the system header files. (For instance, you may have several files that need the definition of *size_t*, but you do not necessarily want to include the whole of each others' contents just to get one definition.)

Inefficiencies occur if two files (for example, *stdio.h* and *stdlib.h*) satisfy the following requirements:

- Both files include the same code segment.
- Both files are included in the given order in one compilation unit, and in the opposite order in another.

This kind of situation creates the problems listed below.

- In the first compilation unit, the precompiled header files for *stdio.h* is built on the condition that *_SIZE_T_DEFINED* was undefined on entry (since the generated precompiled header should contain the definitions of *_SIZE_T_DEFINED* and *size_t*).
- The precompiled file for *stdlib.h* is built on the pre-condition that the macro was defined, and equal to the string "1" (so that the precompiled file has definitions for neither of the two identifiers).

When the second compilation unit first includes *stdlib.h*, it notices that *_SIZE_T_DEFINED* is undefined. The unit must then rebuild the precompiled file for *stdlib.h* so that it can include the definitions for the two identifiers (it has no idea that an include of *stdio.h* is forthcoming, as indeed

it may not). Since it must not contain the definitions of the two identifiers, the precompiled header file for *stdio.h* (which follows) must also be rebuilt.

To avoid this, you should put any such declarations in their own header files (for example, a common one for such common declarations). You should protect these declarations against multiple includes. Each header file that needs a definition of any such common identifier should include this header file.

You must try to minimize the use of #ifs and #ifdefs in header files, except for the outermost multiple-include protection. This is especially important if they are going to be given different values for different compilation units.

Conditions for Not Building Precompiled Headers

Under some conditions, the compiler cannot build a precompiled header file. It builds a stub precompiled header file that records time-stamps and other header information. If such a file is modified or even merely touched, the compiler generates new object code for any compilation unit that includes such a header file.

The factors that prevent the compiler from building a precompiled header file are:

- The include file does not begin or end at file scope. For example, if a file is #included inside a function. This is a problem, since both the file being included and the file doing the #include are marked as “nonprecompilable.”
- If you use templates and exceptions. This release does not implement the mechanisms to represent templates and exception structures in a precompiled header file. If a header file uses or defines a template (that is, a mere direct mention of a template class or function), no precompiled file is created for that header file. However, a reference to an externally defined typedef that expands to a template class instance is permissible.
- If you have a function with a default argument declared in one header file and defined in another header file. The second header file is marked as nonprecompilable.

- If you have a class in one header and in-line constructor or destructor body in another. The second header file is marked as nonprecompilable.
- If you have a static data member declared in one header file and defined in another. The second header file is marked as nonprecompilable.
- If you have an access-adjustment declaration that refers to another access-adjustment declaration in a different header file. The second header file is marked as nonprecompilable.
- If you have explicitly disabled the generation of precompiled files. You can do this with the code

  ```
  #pragma do_not_precompile
  ```

  You can use this as a workaround if you have compiler errors when handling certain header files, but in general, it is not recommended.

Known Problems in the Precompiled Header Mechanism

- If you declare a macro in the top-level compilation unit before a #include that modifies some keyword (or some other identifier never before declared as a macro), the precompiled header mechanism may not rebuild the header file.

  For example, if you have declarations in two different files the compiler fails to recognize that the macro defined in *file2.c* (that redefines *int*) effectively invalidates all the precompiled headers that are included below it. Consider the two examples below:

Example 5-2 file1.c

```
#include “hdr.h”
```

Example 5-3 file2.c

```
#define int long
#include “hdr.h”
```

- If any of the following conditions are true, the compiler may abort.
  - compilation units with templates
  - the compiler automatically including the source files for the template bodies without them being explicitly #included
  - the *-smart* flag is specified

Smart Build Known Problems

This section describes problems in the Smart Build facility.

- Smart Build incorrectly skips recompilation when a macro overrides a non-macro token (for example, a keyword). The precompiled version, using the non-overridden token, will be used. For example, consider the following code:

  ```
  #define protected public
  #include “somefile.h”
  #undef protected
  ```

  If the following conditions are true, Smart Build will fail to notice that this new macro redefines some tokens in the include file and will not rebuild the precompiled header file.
  - The header file *somefile.h* has been previously seen in a build of some other compilation unit (and has thus been already precompiled into a binary file).
  - The word *protected* has not been defined as a macro in any other compilation unit.

  You must compile the unit that contains this unusual *#define* without the *-smart* option.

- Smart Build occasionally skips a recompilation even though a Delta-incompatible change has been made. If the following conditions are true, Smart Build will not detect the change, and will not recompile the changes.
  - A derived class references a global variable
  - A new version of a base class then defines a data member with the same name (thus overriding the global — a Delta-incompatible change that should force the derived class members to be recompiled),

  You must delete the object file and recompile to force Smart Build to rebuild, which it will do correctly.

  In most cases when Smart Build incorrectly skips a file that should have been recompiled, the workaround is to remove the object file(s) and recompile.
- When you build executable code from a single source file don't use the *-c* option, Smart Build may create an empty object file, causing the link to fail. Consider the following command:

  ```
  NCC -smart x.c -o x
  ```

  The first compile will work. *NCC* will remove the x.o object file as part of its normal operation.

  When the second compile is done, SmartBuild will assume that the source file doesn't need to be recompiled and touches the object file, creating an empty object file. The linker prints this message:

  ```
  Remark(2): Smart Build: recompilation skipped

  ld:
  Can't have archive/object only 0 bytes long: x.o
  ```

  You must either omit the *-smart* flag, or compile and link in separate steps, keeping the object file in your directory.

Chapter 6

Code Examples

This chapter contains C++ coding examples. It has the following major sections:

- "cfront Compatibility Examples" on page 53 provides you with examples of code that compiled with *cfront*, but are invalid in the Silicon Graphics C++ environment.
- "Delta-Compatible Changes Examples"
- "Delta-Incompatible Changes Examples"

cfront Compatibility Examples

This section contains examples of *cfront*-compatible code that is illegal in the Silicon Graphics C++ environment. (For a complete description of the *cfront* compatibility issues, see "cfront Compatibility" on page 8.) The examples covered are listed below:

- Terminating comment lines with a backslash
- Explicitly declaring member functions
- Using the same identifier for multiple arguments
- Deleting a pointer to a const
- 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

Terminating Comment Lines With a Backslash

If you use a C++-style (/ /) comment line terminated with a backslash, *CC* and *DCC* will (correctly) continue the comment line into the next source line. *OCC* (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 6-1, *CC* and *DCC* considers the two lines following the / / comment to be part of the comment, while *cfront* does not.

Example 6-1 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.

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 *CC* and *DCC* displays the following error message:

```
error(3414): defining an implicitly declared member function
is not allowed alpha::alpha( const alpha &other )
                ^
```

For example, the code in Example 6-2 will not compile.

Example 6-2 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.

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(3352): a pointer to const may not be deleted
   delete ptr; // Error 352.
          ^
```

For example, the code in Example 6-3 will not compile.

Example 6-3 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.

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(3252): argument of type “volatile mytype *” is
   incompatible with parameter of type “mytype *”
   f1( vol ); // Error.
       ^
```

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

Example 6-4 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

When calling an overloaded function with *char** and *long* variables and passing an integral constant 0 smaller than a *long*, you must explicitly cast the argument to either a *char** or a *long*. If you do not, the following error message is displayed:

```
error(3390): more than one instance of constructor
   "UndoEvent::UndoEvent" matches the argument list:
   function func( char *)"
   function func( long)"
   func( (Bool)False );
```

For example, the code in Example 6-5 will not compile.

Example 6-5 Disambiguating Between a char* and a long

```
  void func( char * ){}
  void func( long ){}

typedef unsigned char Bool;

int main()
{
  func( (Bool) 0 ); // Error 3390.
  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(3177): invalid combination of type specifiers
    unsigned unsigned int x; // Error
             ^
```

For example, the code in Example 6-6 will not compile.

Example 6-6 Rejecting Redundant Type Specifiers

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

You must delete the redundant type specifier.

Note: A *long long* data type is supported.

Implicitly Converting a Pointer to a Pointer to a Different Class

You cannot implicitly cast a pointer to a class to be a pointer to a different class, except when the pointer becomes a pointer to a base-class of the original class. Even when both classes inherit from a same base class, a pointer to one of these classes cannot be implicitly cast to be a pointer to the other class. 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 6-7 will not compile.

Example 6-7 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 common base class.

Assigning a Comma Expression Ending in 0 to a Pointer

You cannot assign a comma expression with a rightmost expression of 0 to a pointer. 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 = ( 0, 0 ); // Error.
     ^
```

For example, the code in Example 6-8 will not compile.

Example 6-8 Assigning a 0 to a Pointer

```
void main()
{
  int* p;
  p = ( 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.

Delta-Compatible Changes Examples

This section contains examples that demonstrate the unique capabilities of the *DCC*. (For a complete description of the *cfront* compatibility issues, see “Delta-Compatible Changes” on page 34.) The examples covered are listed below.

- Adding members to a class
- Adding new base classes
- Promoting members

- Overriding functions
- Reordering members

These examples are for dynamic classes only. Non-dynamic classes (which is the default) do not have this facility; all changes are delta-incompatible.

Adding Members to a Class

DCC allows you to add both member functions and variables to a class without forcing the recompilation of any code that uses that class. This is true for public, protected, and private members. For example, you have a class **Alpha** defined as shown in the following code segment:

```
class Alpha
{
  long a;
  long A();
  virtual long VA();
};
```

You are allowed to extend the members of **Alpha** without recompilation. A typical extension is shown in Example 6-9.

Example 6-9 Adding Members to a Class

```
class Alpha
{
  long a;
  long a1;
  long A();
  virtual long VA();
  virtual long VA1();
};
```

Reordering Members

DCC allows you to reorder the members of a class. You can reorder the member variables to make more efficient use of space or to group members by their protection level. This reordering has no effect on the interface being provided by the class. For example, consider the code sample shown below:

```
class Alpha
{
  long a;
  char c;
  long a1;
  short s;
  long A();
};
```

You can reorder the members as shown in Example 6-10.

Example 6-10 Reordering Members

```
class Alpha
{
  long a;
  long a1;
  char c;
  short s;
  long A();
};
```

Adding New Base Classes

DCC allows you to add a new base class to a class that already exists. For example, consider the code sample shown below:

```
class Alpha
{
  long a;
  long a1;
  long A();
  virtual long VA();
  virtual long VA1();
};
```

You are allowed to extend the classes without recompilation. A typical extension is shown in Example 6-11.

Example 6-11 Adding New Base Classes

```
class Gamma
{
  long g;
  long G();
  virtual long VG();
}

class Alpha : public Gamma
{
  long a;
  long a1;
  long A();
  virtual long VA();
  virtual long VA1();
};
```

Class **Alpha** now supports the additional functionality described in class **Gamma** and still supports **Alpha**'s original interface.

Note: You cannot change from single to multiple inheritance. See "Changing From Single to Multiple Inheritance" on page 70 for more information.

Promoting Members

DCC allows you to move functionality from a derived class to a non-virtual base class, so long as its type remains identical and is not an inline function. For example, consider the code sample shown below:

```
class Gamma
{
  long g;
  long G();
  virtual long VG();
}

class Alpha : public Gamma
{
  long a;
```

```
  long a1;
  long A();
  virtual long VA();
  virtual long VA1();
};
```

If **Alpha** is derived from **Gamma**, you are free to move some of the members from **Alpha** into **Gamma**. The new version of **Alpha** still provides a compatible interface. You won't have to worry about how the functionality of **Alpha** is provided, only that it is provided. For example, consider the code sample shown below:

Example 6-12 Promoting Members

```
class Gamma
{
  long g;
  long a1;
  long G();
  virtual long VG();
  virtual long VA1();
}

class Alpha : public Gamma
{
  long a;
  long A();
  virtual long VA();
};
```

a1 and **VA1()** have been promoted from **Alpha** to **Gamma**. You can release the new versions of **Alpha** and **Gamma** without requiring the recompilation of any code that uses either class.

Overriding Functions

DCC allows you to override a function or variable independent of whether it is a member or virtual member of a class. For example, consider the code sample shown below:

```
class Gamma
{
  long g;
```

```
  long a1;
  long G();
  virtual long VG();
  virtual long VA1();
}

class Alpha : public Gamma
{
  long a;
  long A();
  virtual long VA();
};
```

You can change the overrides as shown in Example 6-13.

Example 6-13 Overriding Functions

```
class Gamma
{
  long g;
  long a1;
  long G();
  virtual long VG();
  virtual long VA1();
}

class Alpha : public Gamma
{
  long a;
  long a1;
  long A();
  virtual long VA();
  long G();
  long VG();
};
```

A user of **Alpha** should be unconcerned whether **Alpha** overrides the member function **G()** or the virtual member function **VG()** (originally declared in **Gamma**). The function that is called when invoking **G()** on an instance of **Alpha** changes, but the code still works.

Delta-Incompatible Changes Examples

This section contains examples of class modifications that are not supported by *DCC*. (For a complete description of the unsupported incompatible changes, see “Delta-Incompatible Changes” on page 35.) The examples covered are listed below:

- Changing member declarations
- Adding overloaded functions to a class
- Overriding functions
- Changing from single to multiple inheritance
- Moving members to an enclosing class

Changing Member Declarations

You may not change the declaration of a member in any way (for example, changing a type from *short* to *long*). This includes types of data members, function declarations, inline function bodies, typedef declarations, and enumeration declarations.

Changing the Values of Enumeration Constants

You may not change the values of enumeration constants. For example, consider the code sample shown below:

```
enum color {
  red,
  blue
};
```

An incompatible change is shown in Example 6-14.

Example 6-14 Changing the Values of Enumeration Constants (1)

```
enum color {
  red,
  yellow,
  blue
};
```

Another incompatible change is shown in Example 6-15.

Example 6-15 Changing the Values of Enumeration Constants (2)

```
enum color {
  red = 4,
  blue = 1
};
```

You may add values to an existing enumeration, provided you do not modify the values of any of the existing enumeration constants.

Adding or Removing Member Function Parameters

You may not add or remove parameters from a member function, and you may not change the return type of a function. For example, consider the code sample shown below:

```
class Alpha {
  void f(int x);
};
```

An incompatible change is shown in Example 6-16.

Example 6-16 Adding or Removing Member Function Parameters

```
class Alpha {
  void f(int x, long y);
};
```

You may add a default value to an existing parameter that did not already have one (such as **`int x = 906`**).

Changing Member Function Default Parameters

You may not change default parameters to member functions. For example, consider the code sample shown below:

```
class Alpha {
  void f(int x = 906);
};
```

An incompatible change is shown in Example 6-17.

Example 6-17 Changing Member Function Default Parameters

```
class Alpha {
  void f(int x = 1);
};
```

Adding Overloaded Functions to a Class

You may add an overloaded function to a class only if the function you are adding cannot possibly be called by existing code. For example, consider the code sample shown below:

```
class Alpha
{
  void F(int);
}
```

An incompatible change is shown in Example 6-18.

Example 6-18 Adding Overloaded Functions to a Class

```
class Alpha
{
  void F(int);
  void F(float);
}
```

Adding the function **F(float)** is an incompatible change, since an existing call of **F(1.0)**, which previously invoked **F(int)**, would still do so. This is because *DCC* determines the function's signature at compile time. Only the location of the function's class is left unresolved until link time.

Overriding Functions

There are two unacceptable incompatibilities when you want to override functions:

- overriding a function defined in a base class with a new version in a derived class
- overriding a global object in a base class when any derived classes reference that object.

Base Class/Derived Class

You may not override an inline function defined in a base class with a new version of the function in a derived class. For example, consider the code sample shown below:

```
class Base
{
  int test()
  {
    printf( “Base::test” );
  }
};

class Derived : public Base
{
};

Derived Object;

main()
{
  object.test();
}
```

object.test() invokes **Base::test()**, which is expanded inline.

Assuming these are dynamic classes, an incompatible change is shown in Example 6-19.

Example 6-19 Overriding Functions: Base Class / Derived Class

```
class Base
{
  int test()
  {
    printf( “Base::test” );
  }
};

class Derived : public Base
{
  int test()
  {
    printf( “Derived::test” );
  }
};

Derived Object;

main()
{
  object.test();
}
```

Unless **main()** is recompiled, it will continue to call **Base::test()**.

Base Class/Global Object

A global object may not be overridden in a base class, when any derived classes reference that object. For example, consider the code sample shown below:

```
int global;

class Alpha
{
};

class Gamma : public Alpha
{
```

```
public:
  func();
};

int gamma::func()
{
  return global;
}
```

An incompatible change is shown in Example 6-20.

Example 6-20 Overriding Functions: Base Class / Global Object

```
int global;

class Alpha
{
  int global;
};

int gamma::func()
{
  return global;
}
```

Changing From Single to Multiple Inheritance

You may not add a base class if the derived class already has exactly one base class. For example, consider the code sample shown below:

```
class Gamma
{
  long g;
  long G();
  virtual long VG();
}

class Alpha : public Gamma
{
  long a;
  long A();
  virtual long VA1();
};
```

An incompatible change is shown in Example 6-21.

Example 6-21 Changing From Single to Multiple Inheritance

```
class Gamma
{
  long g;
  long G();
  virtual long VG();
}

class Beta
{
  long b;
  long B();
  virtual long VB();
}

class Alpha : public Gamma, Beta
{
  long a;
  long A();
  virtual long VA1();
};
```

You may add a base class if the derived class has zero, two, or more base classes (in other words, it is already participating in multiple inheritance).

Moving Members to an Enclosing Class

You may not move a member to an enclosing class, or hide a member from an enclosing class. For example, consider the code sample shown below:

```
struct outer {
  static int s1;
  outer();
  struct inner {
    static int s2;
    inner();
  };
};
```

An incompatible change is shown in Example 6-22.

Example 6-22 Moving Members to an Enclosing Class

```
struct outer {
  static int s1;
  static int s2; // moved
  outer();
  struct inner {
    inner();
    static int s1; //new member
  };
};
```

s2 was moved to an enclosing class, and the new *s1* in *inner* hides the *s1* from *outer*. Both of these changes are delta- incompatible, and you must recompile your code.

Chapter 7

Common Pitfalls

This chapter contains the following major sections:

- “Problems Involving C Linkage” on page 73 discusses some problems you may encounter when you link your C++ programs to the C libraries.
- “Problems With Order of Specification of Libraries” on page 74 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 two 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**, and you are not careful about flushing the output buffers, you may see unexpected results. 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
```

If you wish to avoid explicitly flushing the buffer, you may insert the following code before performing any input/output:

```
ios::sync_with_stdio();
```

Chapter 8

Using Templates

This chapter discusses the Silicon Graphics C++ implementation of templates. It compares the Silicon Graphics implementation to those of the Borland® C++ and *cfront* compilers. It contains the following major sections:

- "CC -32 Template Instantiation" on page 77 describes how you perform template instantiation in the 32-bit Silicon Graphics C++ environment.
- "CC -64 Template Instantiation" on page 91 describes how you perform template instantiation in the 64-bit Silicon Graphics C++ environment.
- "How to Transition From cfront" on page 92 describes how a programmer currently using the cfront template instantiation mechanism can transition to the template instantiation scheme used by the new Silicon Graphics C++ compilers.
- "Template Language Support" on page 95 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*.

CC -32 Template Instantiation

This section describes the 32-bit implementation of templates.

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, *CC -32* 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” on page 82 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.

CC -32 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” on page 81.
- 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.

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

*CC -32*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 *CC -32* 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” on page 89 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 — None of the template entities are instantiated. If automatic instantiation is on (in other words, *-prelink*), any template entities that the prelinker instructs the compiler to instantiate are instantiated.

-ptused — Any template entities used in this compilation unit are 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 — Any template entities declared or referenced in the current compilation unit are 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 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" on page 81.)

-no_auto_include Disables implicit inclusion of template implementation files. (See "Implicit Inclusion" on page 81.)

-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 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

CC -32 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” on page 87 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 Silicon Graphics 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" on page 89).
  - The prelinker cannot compile an object file if it was renamed after being compiled. For example, consider the following command line:

    ```
    yacc gram.y CC -32 -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 *CC -32*. *CC -64* currently does not support automatic template instantiation. The *CC -32* 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" on page 83.
- 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.

How to Transition From cfront

If you have compiled your source code with *cfront*, you may have to modify your build scripts to ensure that your templates are instantiated properly. This section discusses how to transition templates from *cfront* to the Silicon Graphics environment.

Mapping Template Options From cfront to CC -32

The *cfront* template-related options, their meaning, and the equivalent *CC -32* options are listed below:

-pta — Instantiates a whole template class rather than only those members that are needed. If you use automatic instantiation, there is no equivalent option for *CC -32*. If you use explicit instantiation, the *-ptall* option performs roughly the same action.

-pte suffix — Uses *suffix* as the standard source suffix instead of *.c*. There is currently no equivalent *CC -32* option. *CC -32* always looks for the following suffixes when looking for a template body to implicitly include: *.c, .C, .cpp, .CPP, .cxx, .CXX, .cc, .c++*.

-ptn — Changes the default instantiation behavior for one-file programs to that of larger programs, where instantiation is broken out separately and the repository updated. One-file programs normally have instantiation optimized so that instantiation is done into the application object itself. There is currently no equivalent *CC -32* option.

One way of approximating this behavior is to compile your file with *-c*, and then link it, instead of compiling and linking in a single step. Another method is to create an empty dummy file, and compile/link your original file and the new dummy file in a single step. For example, you can use the following command line:

```
CC -32 file.c dummy.c
```

-ptrpathname Specifies a repository, with *./ptrepository* as the default. If several repositories are given, only the first is writable, and the default repository is ignored unless explicitly named. There is no equivalent option for *CC -32*. The *cfront* "repositories" contain two kinds of information:

- information about where types and templates are defined
- object files containing template instantiations

The *CC -32* template instantiation mechanism does not use separate object files for template instantiations; all necessary template instantiations are performed in files that are part of the application (or library) being built. Information about which templates are capable of being instantiated by each file are embedded in the object file itself. This means that no repositories are needed. See "What to Do If You Use Object Files From cfront's Repository" and "What to Do If You Use Multiple Repositories" on page 94 for further information.

-pts Splits instantiations into separate object files, with one function per object (including overloaded functions), and all class static data and virtual functions grouped into a single object. There is no equivalent *CC -32* option. You can exercise fine-grained control over exactly which templates are instantiated in each file by using the instantiation pragmas described in "Pragmas for Template Instantiation" on page 87.

-ptv Turns on verbose or verify mode, which displays each phase of instantiation as it occurs, together with the elapsed time in seconds that phase took to complete. You should use this option if you are new to templates. Verbose mode displays the reason an instantiation is done and the exact

CC command used. The *-ptv* option is also supported by *CC -32*, and provides verbose information about the operation of the prelinker. The prelinker indicates which template instantiations are being assigned to which files, and which files are being recompiled.

What to Do If You Use Object Files From cfront's Repository

If you are used to the *cfront* template instantiation mechanism you may sometimes explicitly reference object files in the repository. This is often done when building an archive or a shared library. The general idea is to link a fake main program with a set of object files so as to populate the repository with the necessary template instantiations. The object files that were linked, along with the object files in the repository, are stored in an archive, or linked into a shared library.

cfront users do this to build an archive or library which has no unresolved template references. *CC -32* users who wish to build archives and shared libraries where all template references have been resolved can do the following:

- If you are building a shared library, the *CC -32* driver will automatically run the prelinker on the set of object files being linked into the shared libraries. No further action is necessary on the part of the programmer.
- If an archive is being built, the prelinker needs to be run explicitly on the object files, before invoking *ar*. See "Building Shared Libraries and Archives" on page 89 for information on how to do this.

What to Do If You Use Multiple Repositories

If you use the *cfront* template instantiation mechanism, you may sometimes use multiple repositories. For example, you may have an application which consists of multiple libraries. Each library is built in its own directory, and has its own repository. When you build the library, template functions are not instantiated. When the application is linked against these libraries, the necessary templates are instantiated at link time. The repositories provide enough information about where to find the necessary template declarations and implementations.

CC -32 does not use repositories, and you can use various strategies when linking a set of object files against a set of libraries that contain references to uninstantiated template functions. Some examples are given below:

- If it is possible that all uninstantiated template functions can be instantiated in the object files being linked into the application, the prelinker will do so automatically. However, it is possible that a library uses a template internally, which is never used by the object files being linked into the application. Such templates are not instantiated by the prelinker, resulting in undefined symbols.
- A better strategy is to prelink each library when it is built, so that the main program is not burdened with having to perform these instantiations. One problem occurs if multiple libraries use the same template functions: if each library is prelinked, multiple copies of such functions will be generated. Removal of duplicate functions takes place only in *.o* and *.a* files; shared libraries cannot have any duplicate code removed.

Template Language Support

The language support for templates in the Silicon Graphics C++ environment is more extensive than for *cfront*. Some of the additional template language constructs supported by the Silicon Graphics C++ environment 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.

- *CC* is 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 *CC*, 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, and conventions.

access

The degree of privilege to a member of a class: *public*, *private*, or *protected*.

base class

Parent of a class.

class extension

Adding base classes to 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 set of data 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 classes

Classes whose layout and location are determined at link time.

instantiate

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

members

Either or both *data* and *method*s 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.

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.

Smart Build

An option to the compiler where only those files that must be recompiled are recompiled.

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.

Index

D

E

F

G

P

R

S

T

U

V

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-0704-070.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389