C/C++ Reference Manual

CONCURRENT COMPUTER CORPORATION™

0890497-030

July 2001

Acknowledgment: This manual contains material contributed by Edison Design Group, Inc. Those portions are copyrighted and reproduced with permission.

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original Release -- July 1996	000	PowerMAX OS 3.1
Current Release -- July 2001	030	PowerMAX OS 5.3

Preface

Scope of Manual

This manual is a reference document on Concurrent C/C++, two general-purpose programming languages.

Information in this manual applies to the platforms described in the latest Concurrent Computer Corporation product catalogs.

System manual page (man page) descriptions of programs, system calls and subroutines can be found online.

Syntax Notation

The following notation is used throughout this guide:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms may also appear in *italic*.
`list bold`	User input appears in **`list bold`** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **`list bold`** type.
`list`	Operating system and program output such as prompts and messages and listings of files and programs appears in `list` type.
`[]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments.
`{}`	Braces enclose mutually exclusive choices separated by the pipe (|) character, where one choice must be selected. You do not type the braces with the choice.
`...`	An ellipsis follows an item that can be repeated.
`::=`	This symbol means “is defined as” in Backus-Naur Form (BNF).

Referenced Publications

The following Concurrent publications are referenced in this document:

0890459	Compilation Systems Volume 1 (Tools)
0890460	Compilation Systems Volume 2 (Concepts)

Contents

1 Compilation

1 Compilation

C and C++ are programming languages suitable for systems programming and general applications. C is a relatively low–level language in that it was designed to accommodate the actual architecture of digital computers.

Many of the advantages of assembly language are available to the C and C++ programmer. These include indirect addressing, address arithmetic, bit manipulation, and access to low–level I/O routines and system services.

A wide variety of operators are also included in the language to take advantage of computer instruction sets, such as shift operators that convert directly into shift right/shift left instructions.

C++ is an extension of the C language, although C++ is not strictly a superset of C. Many of the extensions support object-oriented programming. The evolution of C++ has resulted the development of national and international standards for the language.

The Concurrent C/C++ compiler can be invoked as **`ec`** or **`ec++`**, or through the program development environment tools. Concurrent C/C++ consists of:

- An implementation of the language specified by the ANSI C++ standard. For more information on ANSI C++, refer to the following:
  - *The C++ Programming Language, Third Edition* by B. Stroustrup (Addison-Wesley Publishing Company, Reading, Mass.).
  - *The Annotated C++ Reference Manual* (ARM) by M. Ellis and B. Stroustrup (Addison-Wesley Publishing Company, Reading, Mass.).
  - *International Standard for Information Systems--Programming Language C++*, Document No. X3J16/95-0185 by the American National Standards Institute.
- Partial support of the **`cfront`** dialect of C++. (The **`cfront`** dialect and technology were developed by AT&T's UNIX^{TM} Software Operation, then transferred to Unix System Laboratories, Inc., and finally sold to Novell, Inc.)
- A full implementation of the language specified by the ANSI C standard. For more information on ANSI C, refer to *C: A Reference Manual, Second Edition* by S. Harbison and G. Steele (Prentice–Hall, Inc., Englewood Cliffs, N.J.) and *Programming Languages--C*, ISO/IEC 9899:1990 by the International Organization for Standardization. (The ANSI/ISO standard for C was formerly ANSI document X3.159-1989.)
- A full implementation of the language described in *The C Programming Language* by B. Kernighan and D. Ritchie (Prentice–Hall, Inc., Englewood Cliffs, N.J.)
- Extensions documented in Chapter 6 ("Special Features of C++") and architecture–dependencies documented in Appendix B ("Architecture

Dependencies") . Refer to the on-line manual pages for descriptions of system calls and library routines. Refer to the **ec(1)** and **ec++(1)** man pages for an overview of the Concurrent C/C++ compiler and its options.

- A program development environment (PDE) very similar to the one provided with MAXAda and supported by the NightBench tool. This consists of a number of tools for building large projects and controlling template instantiation, compilation options, library management, etc.

The Concurrent C/C++ compiler consists of front end technology developed and licensed by Edison Design Group, Inc., and back end (code generation and optimization) technology developed and owned by Concurrent Computer Corporation.

This manual presents the features, specifics of implementation, and usage of the Concurrent C/C++ compiler. See the preceding sources for general information on C++ and C.

The Concurrent C/C++ programming environment allows high-level program coding and source-level testing of code. The C and C++ languages are implemented for high-level programming, and they contain many control and structuring facilities that greatly simplify the task of algorithm construction. Each tool (e.g., **ec**, **ec++**, **as**, **ld**) can preserve all the information necessary for meaningful symbolic testing at the source level. The ELF object file format is supported. (For more information, see **ec++(1)**. See "Executable and Linking Format (ELF)" in the *Compilation Systems Volume 2* (*Concepts*) manual.) The environment provides utility packages (e.g., **adb**, **dump**) that aid in testing and debugging. NightView^{TM}, Concurrent's source-level, multi-lingual, multi-processor debugger, is also available.

Compilation Phases

The Concurrent C/C++ compiler, **ec/ec++(1)**, is based on Concurrent's Common Code Generator. The steps involved in creating an executable from C/C++ source appear in the following list and in Figure 1-1

1. Create a file containing C/C++ source code. This is typically done in a text editor like **vi(1)** or **emacs(1)**.

2. Invoke the Concurrent C++ compiler, **ec++(1)**, or the Concurrent C compiler, **ec(1)**, with appropriate options and arguments. See the **ec++(1) ec(1)** man pages, "Invoking the Compiler" on page 1-13 and "Environment Variables" on page 6-37 for information on options, arguments, and environment variables available. See Chapter 7 ("Compilation Modes") for information on options that control compilation modes. Some of the possible arguments include: C or C++ source files (generally end with **.c**, **.C**, or **.cpp**), assembly language source files (must end with **.s**), object files (must end with **.o**), and libraries. Unless you provide options

to cut the process short, **cc++** performs all of the following steps.

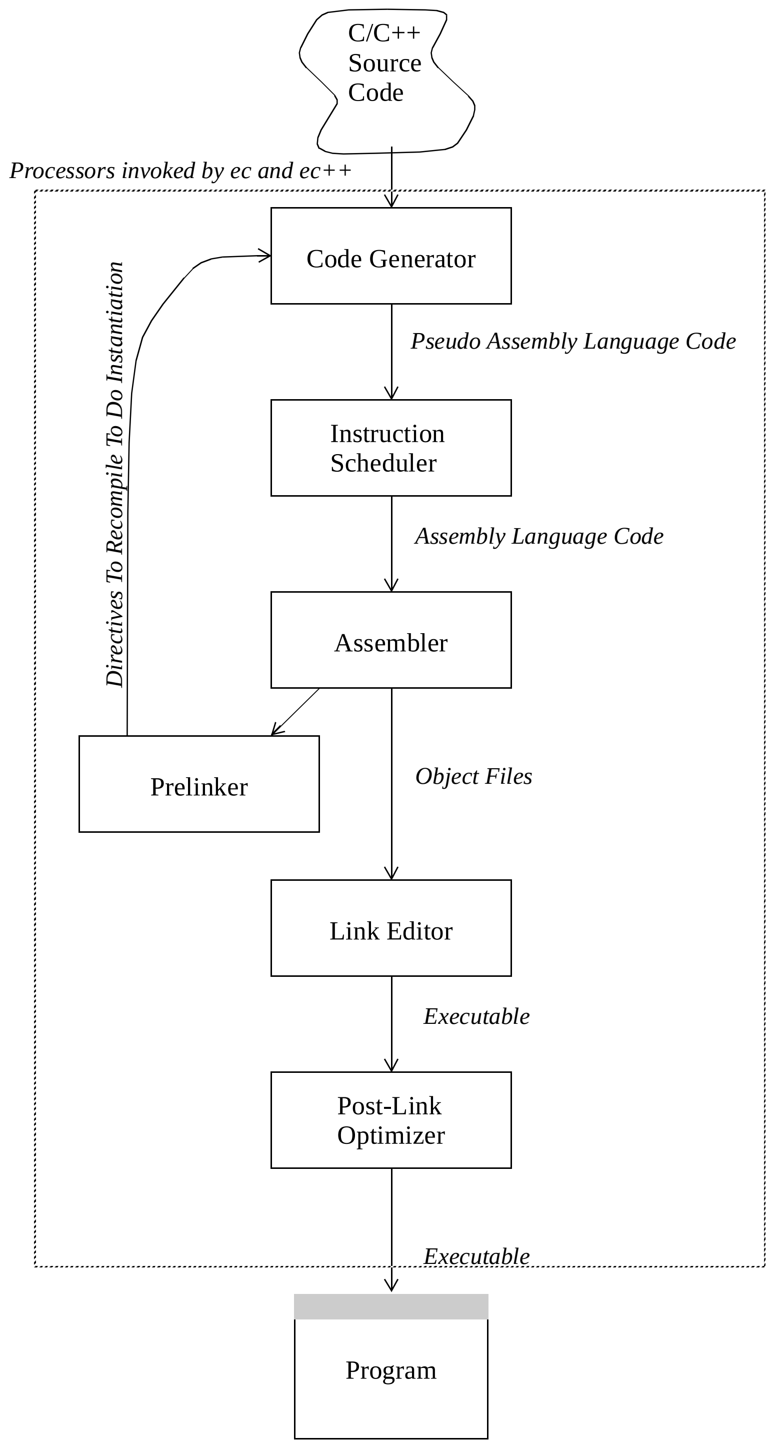

Figure 1-1. Compiling and Linking C++ Programs

A. **ec++** and **ec** call a translator

   /usr/ccs/lib/release/*release***/lib/cxc++**

   to convert the C/C++ source code into pseudo-assembly language. For information on target systems, see “Predefined Macros” on page 6-17 and **cc++(1)**.

B. After producing the pseudo-assembly language code, the compiler calls the instruction scheduler

   /usr/ccs/lib/release/*release***/lib/reorder**

   to perform the final pass of code generation, to schedule instructions, and to translate the pseudo-assembly language code into assembly language.

C. The compiler then calls the assembler, **as(1)**, providing the **.s** files and the output files from **reorder**. The assembler creates object files ending in **.o**. (When object files are created, basenames are retained. For example, if there is a C++ source file named **solver.c**, the name of its object file counterpart is **solver.o**. If there is an assembly language source file named **dynamo.s**, the name of its object file counterpart is **dynamo.o**.)

D. Because automatic instantiation of C++ template entities is not performed in the previous steps, the compiler calls a prelinker

   /usr/ccs/lib/release/*release***/lib/c++prelink**

   to examine object files, looking for information about entities that could be instantiated. See “Template Instantiation” on page 6-3 for a discussion of the procedure used. Note that this procedure may cause files to be recompiled and may generate additional files (**.ti**, **.ii**) to support automatic instantiation., and steps B through D are repeated until there are no more entitites to be instantiated. Auto-instantiation is enabled only if the **--auto_instantiation** option is used or the compilation is being done under control of the Program Development Environment tools.

E. The compiler next calls the link editor, **ld(1)**. The link editor uses two models of linking, static or dynamic. It collects and merges object files and libraries into binary executable load modules.

F. The compiler next calls the post-link optimizer, **analyze(1)**. By default, the executable is named **a.out**. For more information about **analyze**, see “Performance Analysis”, and for more information about **ld**, see “Link Editor and Linking”, both in the *Compilation Systems Volume 1 (Tools)* manual.

Compiler Invocation

The **`ec`** and **`ec++`** compilers accepts many command-line options, also referred to as flags. See the **`ec(1)`** and **`ec++(1)`** man page and Chapter 7 (“Compilation Modes”) for more information. A compiler invocation looks like this:

`$ ec++ [`*options***`]`** *arguments*
`$ ec [`*options***`]`** *arguments*

In the following example, **`part1.c`** and **`part2.c`** are C++ source files, **`part3.s`** is an assembly language source file, and **`part4.o`** is an object file. By default, the compilation and linking is in C++ mode for **`ec++`**, and the compiler automatically performs the steps listed above and creates a binary executable named **`a.out`**.

```
$ ec++ part1.c part2.c part3.s part4.o
```

In the following example, the same files are automatically compiled and linked in strict mode (**`--strict`** option) and the executable is named **`flight_sim`** (**`-o`** option).

```
$ ec++ --strict -oflight_sim part1.c part2.c \
  part3.s art4.o
```

Program Development Environment

Also provided with the Concurrent C/C++ compiler is a high level Program Development Environment (PDE), a set of tools for maintaining complex projects. The PDE maintains a database of all source files, libraries, and executables associated with a defined environment. This database approach has several advantages:

- Concentration of information makes it possible to make queries using tools in the PDE about what options a given object file is built with, what include files were pulled in, etc.
- Template instantiation can deal with libraries better and doesn’t clutter up directories with template and instantiation info files.
- NightBench provides a graphical user interface that sits on top of the PDE, providing the user with an intuitive graphical way of building complex projects.
- A database provides a means for implementing program development tools such as interprocedural analysis and class browsers in future releases of the compilers.

Multiple Release Support

Beginning with release 5.1, the C and C++ compilers support having multiple releases installed at the same time. Additionally, the Concurrent C 4.3 and Concurrent C++ 3.1 compilers can also be installed with C/C++ 5.1.

The follow-on release of both C 4.3 and C++ 3.1 is the C/C++ 5.1 compiler. To access release 5.1 and later, as well as the PDE tools, the user must add **`/usr/ccs/bin`** to his `PATH` environment variable. The C++ compiler is then accessed as **`/usr/ccs/bin/ec++`** and the C compiler is accessed as **`/usr/ccs/bin/ec`** and **`/usr/ccs/bin/ec++ --c`**.

By default, the commands formerly used to invoke C 4.3 (**`/usr/ccs/bin/cc`** and **`/usr/ccs/bin/hc`**) and the commands formerly used to invoke C++ 3.1 (**`/usr/bin/cc++`** and **`/usr/bin/c++`**) will now invoke the default release of C/C++. However, the system administrator can use the **`c.install -p`** option to configure these commands to invoke the pre-5.1 releases by default. Refer to the release notes and “c.install” on page 4-29 for details.

The programs in **`/usr/ccs/bin`** are actually invokers that then invoke the correct release. There is a system wide default release set by the system administrator when he installs the compiler. The user may override that in a number of ways. He may specify a specific release on the command line with the **`--rel`**=*release* option to the compiler (or the **`-rel`** *release* option to the PDE tools), or he may set the environment variable `PDE_RELEASE` to the release he wants, or he may set a user specific default with the **`c.release`** command. The invoker attempts selecting the release by each of these in turn before resorting to the system wide release.

If the PDE is being used to maintain an environment, then the environment remembers what release was used to create it and any tool acting upon that environment will use that release unless the user overrides it with the **`-rel`** option.

The **`c.release`** command can also be used to obtain a list of installed releases. See “c.release” on page 4-59.

Using Concurrent C/C++ with the PLDE

The following should be taken into consideration in order to use Concurrent C/C++ with the PowerWorks Linux Development Environment.

PATH Considerations

On Linux systems, the **`cc`**, **`c++`**, **`gcc`**, and **`g++`** commands invoke the native Linux compilers, which are completely unrelated (and incompatible at the object level) with PowerMAX OS™ and the Concurrent C/C++ cross-compiler.

To utilize the Concurrent C/C++ compiler, specify the following in your PATH environment variable:

```
PATH=$PATH:/usr/ccs/bin
```

The compiler should then be invoked with either **`ec`** or **`ec++`**.

However, if you wish to be able to invoke the Concurrent C/C++ compiler as **`cc`** or **`c++`**, insert the following at the head of your PATH environment variable:

```
PATH=/usr/ccs/crossbin:$PATH
```

The **`/usr/ccs/crossbin`** directory contains commands named **`cc`** and **`c++`** which invoke the Concurrent C/C++ compiler as opposed to the Linux compilers. This directory also contains commands named **`ld`**, **`as`**, and **`nm`**. These will invoke cross versions of these tools rather than the native Linux ones.

See “Makefile Considerations” on page 1-11 for more information.

Include Files and Libraries

By default, the Concurrent C/C++ compiler automatically looks for PowerMAX OS include files and libraries in the tree rooted as:

`/pmax/os/`*version*`/`*arch*

where *version* and *arch* indicate the PowerMAX OS version and target architecture of your choice (see “OS Versions and Target Architectures” on page 1-8 for more details).

Files located under **`/usr/include`** and **`/usr/lib`** are native Linux files and are unrelated and incompatible with the corresponding files for PowerMAX OS. Do not attempt to utilize files from those directories when building PowerMAX OS programs.

Remove any explicit references to these directories in:

- source files (e.g. `#include "/usr/include/unistd.h"`)
- Makefiles (e.g. **`cc -I/usr/include`**)
- build scripts.

Include file references of the form:

```
#include <unistd.h>
```

or

```
#include "unistd.h"
```

need not be changed. These forms are supported, as the appropriate

`/pmax/os/`*version***`/`***arch*

trees are searched.

OS Versions and Target Architectures

The PowerWorks Linux Development Environment supports building PowerMAX OS programs for various versions of PowerMAX OS and various systems.

The current versions of PowerMAX OS (**`osversion`**) that are supported are:

- **`4.3`**
- **`5.0`**

The current architectures (**`arch`**) that are supported are:

- **`nh`**
- **`moto`**
- **`synergy`**

which correspond to the following systems:

Table 1-1. Target Architectures

System type	*architecture*
PowerMAXION-4	**`nh`**
PowerMAXION	**`nh`**
Night Hawk 6800	**`nh`**
Night Hawk 6800 Plus	**`nh`**
TurboHawk	**`nh`**
Power Hawk 610	**`moto`**
Power Hawk 620	**`moto`**
Power Hawk 640	**`moto`**
PowerStack	**`moto`**
PowerStack II	**`moto`**
Power Hawk 710	**`synergy`**
Power Hawk 720	**`synergy`**
Power Hawk 740	**`synergy`**

NOTE

The default OS version is currently **`4.3`** and the default target architecture is **`nh`**.

You can change the **`osversion`** and **`arch`** settings in several ways:

- Specify the options on the **ec** or **ec++** command line:

  ec -o main main.c --arch=*arch* **--osversion=***os*

- Change the default for your user on a specific Linux system using the Concurrent C/C++ command line utility **c.release**:

  c.release -arch *arch* **-osversion** *os*

When using the Concurrent C/C++ PDE utilities (**c.build**, etc.), you can:

- Set the **arch** and **osversion** for an environment using **c.mkenv**:

  c.mkenv -arch *arch* **-osversion** *os*

- Set the **arch** and **osversion** for a specific partition using **c.partition**:

  c.partition -oset "--arch=*arch* **--osversion=os" main**

- Set the **arch** and **osversion** for a specific compilation unit using **c.options**:

  c.options -set -- --arch=*arch* **--osversion=***os* **main**

NOTE

The **arch** and **osversion** selects the `include` files used during compilation and the libraries used during linking. The user should insure that they are specified both when compiling and linking with **ec** and **ec++**, and are specified for both units and partitions when using **c.options** and **c.partition**. The **c.mkenv** command illustrated above sets both the default compile and default link options at the same time.

Shared vs. Static Linking

By default, the Concurrent C/C++ compiler links with shared libraries. Thus, if you attempt to execute your C++ program on a PowerMAX OS system it will require, at a minimum, the shared library **libCruntime.so** or **libCruntime_mt.so**.

If your PowerMAX OS system doesn't have either the Concurrent C/C++ product or the c++runtime package installed, your program will fail to execute. You can install the full PowerMAX OS version of the Concurrent C/C++ compiler, install just the c++runtime package, or relink your program using static libraries.

The PowerMAX OS c++runtime package is included on the PowerWorks Linux Development Environment Installation CD. See the section titled “Target Installation” in the PowerWorks Linux Development Environment Release Notes (0898000) for installation instructions.

To link your program using static libraries, append the **-Zlink=static** option to your command line:

```
ec++ -o main main.c -Zlink=static
```

Or specify the **`-Zlink=static`** option on an executable partition using the **`c.partition`** command:

```
c.partition -oset “-Zlink=static” main
```

Use the **`-Zlibs`** option to control whether individual libraries specifed by the **`-l`** option are dynamically or statically linked (if both are available). For example:

```
ec++ main.c -Zlink=dynamic -Zlibs=static \
            -lposix9 -Zlibs=dynamic -lnsl”
```

will force linking with the archive **`libposix9.a`** while linking with the shared object file **`libnsl.so`**.

Two libraries are implicitly linked with, the C library, **`libC`**, and the C++ library, **`libCruntime`**. You must link with the shared object version of **`libC`** if a program is dynamically linked. To force linking with the archive version of **`libCruntime`**, use the **`--static_Cruntime`** option. Other methods may work or have worked in the past, but are not guaranteed to continue working in the future.

Makefile Considerations

Makefiles may already contain references to **`cc`** or **`c++`** commands explicitly within them. Additionally, if default rules for compilation, such as

```
.c.o:
```

or

```
.cc.o:
```

are not explicitly mentioned, the make processor will also attempt to invoke **`cc`**, **`c++`**, or even **`g++`**.

By default, unless you have **`/usr/ccs/crossbin`** early in your `PATH` variable, these situations will result in the Linux native compilers being invoked instead of the Concurrent C/C++ compiler.

To resolve these problems you can take any of the following approaches.

Explicit Modification Using ec/ec++

Ensure that **`/usr/ccs/bin`** is in your `PATH` environment variable.

Modify all occurrences of **`cc`** and **`c++`** to utilize **`ec`** and **`ec++`**, respectively.

Supply default **`.c.o`** rules (and the like) to explicitly utilize the **`ec`** and **`ec++`** commands.

Use of /usr/ccs/crossbin in PATH Environment Variable

Put **`/usr/ccs/crossbin`** at the head of your `PATH` environment variable.

This will cause references to **`cc`** and **`c++`** to invoke the Concurrent C/C++ compiler as opposed to the Linux compilers.

Use of CC Environment or Make Variables

If you don't want **`/usr/ccs/crossbin`** early on your `PATH` (perhaps because you plan to build for Linux and/or PowerMAX OS at various times), then you'll want to just use the **`ec`** and **`ec++`** when you want to compile for PowerMAX OS (it is still necessary to add **`/usr/ccs/bin`** to your `PATH`).

One approach to using **`ec`** and **`ec++`** that requires minimal changes to Makefiles, etc., is to use environment variables or **`make`** variables to control which C/C++ compiler you're using. The following commands will all build using the PLDE cross-compilers:

Short-lived environment variables:

```
$ CC=ec CXX=ec++ make arguments
```

make variables:

```
$ make arguments CC=ec CXX=ec++
```

Long-lived environment variables:

```
$ export CC=ec
$ export CXX=ec++
$ make arguments
```

You can also use the long-lived environment variable approach if you intend to always build for PowerMAX OS, by adding the following to your login script (e.g. **.profile** or **.login** depending on your shell):

```
export CC=ec
export CXX=ec++
```

Or, if you prefer finer-grained control, you can add lines like the following to the top of any Makefiles that should use the Concurrent C/C++ cross-compiler:

```
CC=ec
CXX=ec++
```

The changes will then only affect the modified Makefiles. Note that this solution only works for Makefiles that use the default **.c.o** and **.cpp.o**, etc. rules. If they contain hard-coded references to **cc** or **cc++**, then either **/usr/ccs/crossbin** must be used, or the Makefiles must be changed to use **$(CC)** and **$(CXX)** instead. If the Makefile references anything like **g++** (Linux's GNU C++ compiler), then it will need to be changed, regardless.

Here are two more complete and robust sets of variables which will work equally well with well-written Makefiles.

```
CC=/usr/ccs/crossbin/cc
CXX=/usr/ccs/crossbin/c++
AS=/usr/ccs/crossbin/as
AR=/usr/ccs/crossbin/ar
LD=/usr/ccs/crossbin/ld
```

Or, alternatively:

```
CC=/usr/ccs/bin/ec
CXX=/usr/ccs/bin/ec++
AS=/usr/ccs/bin/as.pmax
AR=/usr/ccs/bin/ar.pmax
LD=/usr/ccs/bin/ld.pmax
```

These two sets are mentioned in order to provide very easy support for those users that want to compile only for PowerMAX OS (**/usr/ccs/crossbin**) and for those users that may want to compile for either Linux or PowerMAX OS, depending on the application (**/usr/ccs/bin**).

Invoking the Compiler

The compiler is invoked by a command of the form

> **`ec[++] [`***options***`]`** *ifile*

to compile the single input file *ifile*. If **`-`** (hyphen) is specified for *ifile*, the compiler reads from **`stdin`**.[1]

Command line options may be specified using either single character option codes (e.g., **`-o`**) or keyword options (e.g., **`--output`**). A single character option specification consists of a hyphen followed by one or more option characters (e.g., **`-Ab`**). If an option requires an argument, the argument may immediately follow the option letter, or may be separated from the option letter by white space. A keyword option specification consists of two hyphens followed by the option keyword (e.g., **`--strict`**). If an option requires an argument, the argument may be separated from the keyword by white space, or the keyword may be immediately followed by **`=`***option*. When the second form is used there may not be any white space on either side of the equals sign.

A list of files may appear for *ifile*. If a list of files is specified, options that specify a compilation output file (**`--output`**, **`--list`**, and **`--xref`**) may not be used, and the name of each source file is written to **`stderr`** as the compilation of that file begins.

When one of the preprocessing-only modes is specified (see below), the **`--output`** option can be used to specify the preprocessing output file. If **`--output`** is not specified, preprocessing output is written to **`stdout`**. Preprocessing output has trigraphs and line splices processed (and thus they do not appear in their original form).

When compilation (rather than just preprocessing) is done, the output (if any) from the compilation is written to a file selected by the back end; see the documentation of the back end for further information. For versions of the front end that generate an intermediate language file, the **`--output`** option can be used to specify the IL output file.

1. This is not recommended in general, since diagnostic messages and the like will then not include a file name or will refer to the file name "**`-`**".

Command Line Options

The *options* to **`ec[++]`** are broken down into the following categories:

- Compilation Process (see page 1-14)
- Language Dialect (see page 1-24)
- Optimization (see page 1-34)
- Linking (see page 1-41)

Compilation Process

The *options* to **`ec[++]`** concerned with controlling the compilation process are broken down into the following categories:

- Preprocessing (see page 1-14)
- C++ Specific Features (see page 1-18)
- Error Messages (see page 1-19)
- Other (see page 1-20)

Preprocessing

The following are **`ec[++]`** *options* related to preprocessing**:**

`--preprocess`
`-E`

Do preprocessing only. Write preprocessed text to the preprocessing output file, with comments removed and with line control information.

`--no_line_commands`
`--preprocess_to_file`
`-P`

Do preprocessing only. Write preprocessed text to the preprocessing output file, with comments removed and without line control information.

`--comments`
`-C`

Keep comments in the preprocessed output. This should be specified after either **`--preprocess`** or **`--no_line_commands`**; it does not of itself request preprocessing output.

`--dependencies`

Do preprocessing only. Instead of the normal preprocessing output, generate on the preprocessing output file a list of dependency lines suitable for input to the UNIX® **`make`** program. Note that when implicit inclusion of templates is enabled, the output may indicate false (but safe) dependencies.

`--trace_includes`

Do preprocessing only. Instead of the normal preprocessing output, generate on the preprocessing output file a list of the names of files `#include`d.

`--define_macro` *name* **`[(`***parm-list***`)] [=`** *def***`]`**
`-D` *name* **`[(`***parm-list***`)] [=`** *def***`]`**

Define macro *name* as *def*. If "`=` *def*" is omitted, define *name* as `1`. Function-style macros can be defined by appending a macro parameter list to *name*.

`--undefine_macro` *name*
`-U`*name*

Remove any initial definition of the macro *name*. **`--undefine_macro`** options are processed after all **`--define_macro`** options in the command line have been processed.

`--include_directory` *dir*
`--sys_include` *dir*
`-I`*dir*

Add *dir* to the list of directories searched for `#include`s. See "Finding Include Files" on page 6-40.

`--incl_suffixes` *str*

Specifies the list of suffixes to be used when searching for an include file whose name was specified without a suffix. The argument is a colon-separated list of suffixes (e.g., "h:hpp::"). If a null suffix is to be allowed, it must be included in the suffix list.

The default value is "::h:hpp".

`--preinclude` *filename*
`-i`*filename*

Include the source code of the indicated file at the beginning of the compilation. This can be used to establish standard macro definitions, etc. The file name is searched for in the directories on the include search list.

`--list` *lfile*

Generate raw listing information in the file *lfile*. This information is likely to be used to generate a formatted listing. The raw listing file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the front end.

Each line of the listing file begins with a key character that identifies the type of line as described in "Key Characters" on page 1-16.

`--pch`

Automatically use and/or create a precompiled header file. If **`--use_pch`** or **`--create_pch`** (manual PCH mode) appears on the command line following this option, its effect is erased.

`--create_pch` *filename*

If other conditions are satisfied, create a precompiled header file with the specified name. If **`--pch`** (automatic PCH mode) or **`--use_pch`** appears on the command line following this option, its effect is erased.

`--use_pch` *filename*

Use a precompiled header file of the specified name as part of the current compilation. If **`--pch`** (automatic PCH mode) or **`--create_pch`** appears on the command line following this option, its effect is erased.

`--pch_dir` *directory-name*

The directory in which to search for and/or create a precompiled header file. This option may be used with automatic PCH mode (**`--pch`**) or with manual PCH mode (**`--create_pch`** or **`--use_pch`**).

`--pch_messages`
`--no_pch_messages`

Enable or disable the display of a message indicating that a precompiled header file was created or used in the current compilation.

`--list_macros`

List all macro definitions to **`stdout`**.

`--pch_verbose`

In automatic PCH mode, for each precompiled header file that cannot be used for the current compilation, a message is displayed giving the reason that the file cannot be used.

Key Characters

When the **`--list`** *lfile* option is used, the generated raw listing file (*lfile*) contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the front end.

Each line of the listing file begins with a key character that identifies the type of line, as follows:

`N:`

a normal line of source; the rest of the line is the text of the line.

`X:`

the expanded form of a normal line of source; the rest of the line is the text of the line. This line appears following the `N` line, and only if the line contains non-trivial modifications (comments are considered trivial modifications; macro expansions, line splices, and trigraphs are considered non-trivial modifications).

`S`:

a line of source skipped by an `#if` or the like; the rest of the line is text. Note that the `#else`, `#elif`, or `#endif` that ends a skip is marked with an `N`.

`L`:

an indication of a change in source position. The line has a format similar to the # line-identifying directive output by **`cpp`**, that is to say

`L` *line-number filename key*

where *key* is

`1`	for entry into an include file
`2`	for exit from an include file

and omitted otherwise.

The first line in the raw listing file is always an `L` line identifying the primary input file. `L` lines are also output for `#line` directives (*key* is omitted). `L` lines indicate the source position of the following source line in the raw listing file.

`R`, `W`, `E`, or `C`:

an indication of a diagnostic

The line has the form

S filename line-number column-number message-text

where *S* is:

`R`	remark
`W`	warning
`E`	error
`C`	catastrophic error

Errors at the end of file indicate the last line of the primary source file and a column number of zero. Command-line errors are catastrophes with an empty file name (`""`) and a line and column number of zero. Internal errors are catastrophes with position information as usual, and message-text beginning with `(internal error)`. When a diagnostic displays a list (e.g., all the contending routines when there is ambiguity on an overloaded call), the initial diagnostic line is followed by one or more lines with the same overall format (code letter, file name, line number, column number, and message text), but in which the code letter is the lowercase version of the code letter in the initial line. The source position in such lines is the same as that in the corresponding initial line.

C++ Specific Features

The following are **`ec[++]`** *options* related to C++ specific features:

`--auto_instantiation`
`--no_auto_instantiation`

Enable or disable automatic instantiation of templates. This option is valid only in C++ mode.

The default is **`--no_auto_instantiation`** unless the compilation is done under control of the PDE.

`--one_instantiation_per_object`

Put out each template instantiation in this compilation (function or static data member) in a separate object file. The primary object file contains everything else in the compilation, i.e., everything that isn't an instantiation. Having each instantiation in a separate object file is very useful when creating libraries, because it allows the user of the library to pull in only the instantiations that are needed. That can be essential if two different libraries include some of the same instantiations. This option is valid only in C++ mode.

`--instantiation_dir` *dir-name*

When **`--one_instantiation_per_object`** is used, this option can be used to specify a directory into which the generated object files should be put. The default is `Template.dir` if the option is not specified by the user. If the compilation is done under control of the PDE tools, then the object files are kept in the PDE's unit cache.

`--implicit_include`
`--no_implicit_include`

Enable or disable implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. This option is valid only in C++ mode.

The default is **`--no_implicit_include`**.

`--pending_instantiations=`*n*

Specifies the maximum number of instantiations of a given template that may be in process of being instantiated at a given time. This is used to detect runaway recursive instantiations. If *n* is zero, there is no limit.

The default is 64.

`--retain_out_of_line_copy`
`-Qretain_out_of_line_copy`

Retain an out-of-line copy of inlined functions, even if not needed because it gets called or its address gets taken. This is on by default when the **`-g`** option is used.

`--suppress_vtbl`

Suppress definition of virtual function tables in cases where the heuristic used by the front end to decide on definition of virtual function tables provides no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity). The option **`--suppress_vtbl`** suppresses the definition of the virtual function tables for such classes, and **`--force_vtbl`** forces the definition of the virtual function table for such classes. **`--force_vtbl`** differs from the default behavior in that it does not force the definition to be local. This option is valid only in C++ mode.

`--force_vtbl`

Force definition of virtual function tables in cases where the heuristic used by the front end to decide on definition of virtual function tables provides no guidance. See **`--suppress_vtbl`**. This option is valid only in C++ mode.

`--instantiate` *mode*

Control instantiation of external template entities. External template entities are external (i.e., noninline and nonstatic) template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition):

`none`	Instantiate no template entities. This is the default.
`used`	Instantiate only the template entities that are used in this compilation.
`all`	Instantiate all template entities whether or not they are used.
`local`	Instantiate only the template entities that are used in this compilation, and force those entities to be local to this compilation.

This option is valid only in C++ mode.

Error Messages

The following are **`ec[++]`** *options* related to error messages:

`--no_warnings`
`-w`

Suppress warnings. Errors are still issued.

`--remarks`
`--nitpick`
`-n`

Issue remarks, which are diagnostic messages even milder than warnings.

`--error_limit` *number*
`-e` *number*

Set the error limit to *number*. The front end will abandon compilation after this number of errors (remarks and warnings are not counted toward the limit). By default, the limit is 100.

`--diag_suppress` *tag*, *tag*,...
`--diag_remark` *tag*, *tag*,...
`--diag_warning` *tag*, *tag*,...
`--diag_error` *tag*, *tag*,...

Override the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag or using an error number.

`--display_error_number`

Display the error message number in any diagnostic messages that are generated. The option may be used to determine the error number to be used when overriding the severity of a diagnostic message.

`--no_use_before_set_warnings`

Suppress warnings on local automatic variables that are used before their values are set. The front end's algorithm for detecting such uses is conservative and is likely to miss some cases that an optimizer with sophisticated flow analysis could detect; thus, a user might choose to suppress the warnings from the front end when optimization has been requested but to permit them when the optimizer is not being run.

`--error_output` *efile*

Redirect the output that would normally go to **`stderr`** (i.e., diagnostic messages) to the file *efile*. This option is useful on systems where output redirection of files is not well supported. If used, this option should probably be specified first in the command line, since otherwise any command-line errors for options preceding the **`--error_output`** would be written to **`stderr`** before redirection.

`--brief_diagnostics`
`--no_brief_diagnostics`

Enable or disable a mode in which a shorter form of the diagnostic output is used. When enabled, the original source line is not displayed and the error message text is not wrapped when too long to fit on a single line.

`--wrap_diagnostics`
`--no_wrap_diagnostics`

Enable or disable a mode in which the error message text is not wrapped when too long to fit on a single line.

Other

The following are miscellaneous **`ec[++]`** *options* concerned with controlling the compilation process:

`--output` *ofile*
`-o` *ofile*

Specify the output file of the compilation, i.e., the preprocessing or intermediate language output file.

`--version`

Display the version number.

`--no_code_gen`
`-k`

Do syntax-checking only, i.e., do not run the back end.

`--xref` *xfile*

Generate cross-reference information in the file *xfile*. For each reference to an identifier in the source program, a line of the form:

symbol-id name ref-code filename line-number column-number

is written, where *ref-code* is

`D`	definition
`d`	declaration (that is, a declaration that is not a definition)
`M`	modification
`A`	address taken
`U`	used
`C`	changed (but actually meaning “used and modified in a single operation,” such as an increment)
`R`	any other kind of reference
`E`	an error in which the kind of reference is indeterminate

symbol-id is a unique decimal number for the symbol. The fields of the above line are separated by tab characters.

`--timing`

Generate compilation timing information. This option causes the compiler to display the amount of CPU time and elapsed time used by each phase of the compilation and a total for the entire compilation.

`--remove_unneeded_entities`
`--no_remove_unneeded_entities`

Enable or disable an optimization to prune the IL tree of types, variables, routines, and related IL entries that are not “really needed.” (Something may be referenced but unneeded if it is referenced only by something that is itself unneeded; certain entities, such as global variables and routines defined in the translation unit, are always considered to be needed.)

`--debug`
`-g`

Produce additional symbolic debugging information for use with NightView.

--full_debug_info
-Qfull_debug_info

Generate debugging information for every entity declared in a compilation unit. Normally debugging information is created only for types that are actually used in the compilation unit.

--help
--help_screen
-H

Display a help message showing invocation options for this compiler.

--leave_temp_files
-Qleave_temp_files

Do not remove the intermediate files created during compilation.

--symtab_size=*symtab_size*
-T*symtab_size*

Passed to **as(1)**.

--verbose
-v

Be verbose when running the compiler. This option causes informational messages about compilation and optimization to be written to **stderr**. This information can be used to determine the processors (and their arguments) invoked during the compilation.

-S

Compile the named files and leave the assembler-language output in the corresponding files suffixed by **.s**. No object file or executable is produced. (See also **-c**.)

-c

Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled. (See also **-S**.)

--bin_path=*string*
-b*string*

Search for alternative assembler, link editor, and post-link optimizer processors. The compiler removes the path prefix from the default processor and uses string as a substitute prefix for the default processor basename. For example, if the compiler is invoked with **-b/abc/**, the compiler searches for an assembler named **/abc/as**. If the compiler is invoked with **-b/dev/test_**, the compiler searches for an assembler named **/dev/test_as**. Multiple **-b** options may be used to specify multiple strings to try.

--lib_path=*string*
-B*string*

Search for alternative compiler, instruction scheduler, startup routines, and auxiliary object files. The compiler removes the path prefix from the default processor and uses *string* as a substitute prefix for the default processor basename. For example, if **`ec++`** is invoked with **`-B/abc/`**, it searches for a compiler named **`/abc/cxc++`**. Multiple **`-B`** options may be used to specify multiple strings to try.

`--processors=[craPlAs]`
`-t[craPlAs]`

Find only the designated compiler passes using the paths specified with a **`-B`** or **`-b`** option. The letters indicate processors as follows:

`c`	compiler
`r`	reorder, also known as instruction scheduler
`a`	assembler
`P`	prelinker
`l`	link editor
`A`	analyze, also known as post link optimizer
`s`	startup routines and auxiliary object files

`--pass_to_analyze=`*arg1***`[,`***arg2***`...]`**
`--pass_to_assembler=`*arg1***`[,`***arg2***`...]`**
`--pass_to_code_generator=`*arg1***`[,`***arg2***`...]`**
`--pass_to_front_end=`*arg1***`[,`***arg2***`...]`**
`--pass_to_linker=`*arg1***`[,`***arg2***`...]`**
`--pass_to_prelink=`*arg1***`[,`***arg2***`...]`**
`--pass_to_prelinker=`*arg1***`[,`***arg2***`...]`**
`--pass_to_reorder=`“*options***”**
`-W`*x***`,`***arg1***`[,`***arg2***`...]`**

hand off the specified arguments to the processor *x*, where *x* is the letter corresponding to the appropriate processor:

`c`	compiler
`r`	reorder, also known as instruction scheduler
`a`	assembler
`P`	prelinker
`l`	link editor
`A`	analyze, also known as post link optimizer
`s`	startup routines and auxiliary object files

This can be used to specify special arguments to particular processors that the compiler invokes during compilation.

An alternative method is the setting of environment variables:

```
PATH_TO_MCRT0
PATH_TO_CRT0
PATH_TO_STRICT
PATH_TO_ANSI
```

allow the user to specify alternative startup routines and auxiliary object files

```
PATH_TO_CXCPP
PATH_TO_REORDER
PATH_TO_AS
PATH_TO_LD
PATH_TO_ANALYZE
PATH_TO_DECODE
```

allow the user to specify alternative **cxc++**, **reorder**, **as**, **ld**, **analyze**, and **c++decode** tools, respectively.

--limit_search_paths
-X

Do not look in unspecified search paths for include files or compilation processors. An error message will be generated if the files cannot be found in the specified search paths

--cfront_io
--no_cfront_io

Enable or disable automatic link and prelink inclusion of the cfront `<iostream.h>` based **-lCio** archive/library, or for threaded applications the **-lCio_mt** archive/library. This is disabled by default unless either **--cfront_2.1** or **--cfront_3.0** is specified.

--rel=*release*

Select which release of the compiler (post-5.1) to invoke.

--testing
-#

Don't actually do anything. Use with **-v** option to see what the compiler would invoke.

Language Dialect

The following are **ec[++]** *options* related to language dialect:

--c++

Enable compilation of C++. This is the default for **ec++**.

--c

Enable compilation of C rather than C++. This is the default for **ec**.

--old_c
-Xo

Enable K&R/pcc mode, which approximates the behavior of the standard UNIX **pcc**. ANSI C features that do not conflict with K&R/pcc features are still supported in this mode.

-Xa

Enable ANSI C mode. This is the default mode when C mode is selected.

-Xc
--strict_warnings
--strict

Enable strict ANSI mode, which provides diagnostic messages when non-ANSI features are used, and disables features that conflict with ANSI C or C++. This is compatible with both C and C++ mode (although ANSI conformance with C++ does not yet mean anything). It is not compatible with pcc mode. ANSI violations can be issued as either warnings or errors depending on which command line option is used. The **--strict** option causes errors to be issued whereas the **--strict_warnings** and **-Xc** options produce warnings. The error threshold is set so that the requested diagnostics will be listed.

--anachronisms
--no_anachronisms

Enable or disable anachronisms in C++ mode. This option is valid only in C++ mode.

The default is **--no_anachronisms**.

--cfront_2.1

Enable compilation of C++ with compatibility with **cfront** version 2.1. This causes the compiler to accept language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (**cfront**) release 2.1. This option also enables acceptance of anachronisms.

--cfront_3.0

Enable compilation of C++ with compatibility with **cfront** version 3.0. This causes the compiler to accept language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (**cfront**) release 3.0. This option also enables acceptance of anachronisms.

--signed_chars
-Qchars_signed
-Qsigned_char

Make plain `char` signed. The default "signedness" for `char` is unsigned, as this is more efficient on the PowerPC architecture. When plain `char` is signed, the macro `__SIGNED_CHARS__` is defined by the front end.

--unsigned_chars

Make plain `char` unsigned.

--distinct_template_signatures
--no_distinct_template_signatures

Control whether the signatures for template functions can match those for non-template functions when the functions appear in different compilation units. The default is **--distinct_template_signatures**, under which a normal function cannot be used to satisfy the need for a template instance; e.g, a function "`void f(int)`" could not be used to satisfy the need for an instantiation of a template "`void f(T)`" with `T` set to `int`.

--no_distinct_template_signatures provides the older language behavior, under which a non-template function can match a template function. Also controls whether function templates may have template parameters that are not used in the function signature of the function template.

--nonstd_qualifier_deduction
--no_nonstd_qualifier_deduction

Controls whether nonstandard template argument deduction should be performed in the qualifier portion of a qualified name. With this feature enabled, a template argument for the template parameter `T` can de deduced in contexts like `A<T>::B` or `T::B`. The standard deduction mechanism treats these as nondeduced contexts that use the values of template parameters that were either explicitly specified or deduced elsewhere.

--exceptions
--no_exceptions

Enable or disable support for exception handling. This option is valid only in C++ mode.

The default is **--exceptions**.

--rtti
--no_rtti

Enable or disable support for RTTI (runtime type information) features: `dynamic_cast`, `typeid`. This option is valid only in C++ mode.

The default is **--rtti**.

--array_new_and_delete
--no_array_new_and_delete

Enable or disable support for array new and delete. This option is valid only in C++ mode.

`--explicit`
`--no_explicit`

Enable or disable support for the `explicit` specifier on constructor declarations. This option is valid only in C++ mode.

The default is **`--explicit`**.

`--namespaces`
`--no_namespaces`

Enable or disable support for namespaces. This option is valid only in C++ mode.

The default is **`--namespaces`**.

`--old_for_init`
`--new_for_init`

Control the scope of a declaration in a `for-init` statement. The old (**`cfront`**-compatible) scoping rules mean the declaration is in the scope to which the `for` statement itself belongs; the new (standard-conforming) rules in effect wrap the entire `for` statement in its own implicitly generated scope. This option is valid only in C++ mode.

The default is **`--new_for_init`**.

`--for_init_diff_warning`
`--no_for_init_diff_warning`

Enable or disable a warning that is issued when programs compiled under the new `for-init` scoping rules would have had different behavior under the old rules. The diagnostic is only put out when the new rules are used. This option is valid only in C++ mode.

The default is **`--for_init_diff_warnings`**.

`--old_specializations`
`--no_old_specializations`

Enable or disable acceptance of old-style template specializations (i.e., specializations that do not use the `template<>` syntax). This option is valid only in C++ mode.

The default is **`--old_specializations`**.

`--guiding_decls`
`--no_guiding_decls`

Enable or disable recognition of “guiding declarations” of template functions. A guiding declaration is a function declaration that matches an instance of a function template but has no explicit definition (since its definition derives from the function template). For example:

```
template <class T> void f(T) { ... } void f(int);
```

When regarded as a guiding declaration, `f(int)` is an instance of the template; otherwise, it is an independent function for which a definition must be supplied. If

--no_guiding_decls is combined with **--old_specializations**, a specialization of a non-member template function is not recognized — it is treated as a definition of an independent function. This option is valid only in C++ mode.

The default is **--guiding_decls**.

--implicit_extern_c_type_conversion
--no_implicit_extern_c_type_conversion

Enable or disable an extension to permit implicit type conversion in C++ between a pointer to an `extern "C"` function and a pointer to an `extern "C++"` function. (It is useful for **cfront** compatibility — in standard C++ the linkage specification is part of the function type, with the consequence that otherwise identical function types, one declared `extern "C"` and the other declared `extern "C++"`, are viewed as distinct.)

The default is **--implicit_extern_c_type_conversion**.

--long_preserving_rules
--no_long_preserving_rules

Enable or disable the K&R usual arithmetic conversion rules with respect to `long`. This means the rules of K&R I, Appendix A, 6.6, not the rules used by the **pcc** compiler. The significant difference is in the handling of "`long` *op* `unsigned int`" when `int` and `long` are the same size. The ANSI/ISO/pcc rules say the result is `unsigned long`, but K&R I says the result is `long` (`unsigned long` did not exist in K&R I).

--extern_inline
--no_extern_inline

Enable or disable support for inline functions with external linkage in C++. When inline functions are allowed to have external linkage (as required by the standard), then `extern` and `inline` are compatible specifiers on a nonmember function declaration; the default linkage when `inline` appears alone is external (that is, `inline` means `extern inline` on nonmember functions); and an inline member function takes on the linkage of its class (which is usually external). However, when inline functions have only internal linkage (as specified in the ARM), then `extern` and `inline` are incompatible; the default linkage when `inline` appears alone is internal (that is, `inline` means `static inline` on nonmember functions); and inline member functions have internal linkage no matter what the linkage of their class.

--restrict
--no_restrict

Enable or disable recognition of the `restrict` keyword.

--long_lifetime_temps
--short_lifetime_temps

Select the lifetime for temporaries: "short" means to end of full expression; "long" means to the earliest of end of scope, end of switch clause, or the next label.

"short" is standard C++, and "long" is what **cfront** uses (the **cfront** compatibility modes select "long" by default).

`--wchar_t_keyword`
`--no_wchar_t_keyword`

Enable or disable recognition of `wchar_t` as a keyword. This option is valid only in C++ mode.

The default is **`--wchar_t_keyword`**.

`--bool`
`--no_bool`

Enable or disable recognition of `bool`. This option is valid only in C++ mode, or in C mode if the target architecture supports AltiVec enhancements.

The default is **`--bool`**.

`--typename`
`--no_typename`

Enable or disable recognition of `typename`. This option is valid only in C++ mode.

The default is **`--typename`**.

`--implicit_typename`
`--no_implicit_typename`

Enable or disable implicit determination, from context, whether a template parameter dependent name is a type or nontype. This option is valid only in C++ mode.

The default is **`--implicit_typename`**.

`--special_subscript_cost`
`--no_special_subscript_cost`

Enable or disable a special nonstandard weighting of the conversion to the integral operand of the `[]` operator in overload resolution. This is a compatibility feature that may be useful with some existing code. The special cost is enabled by default in **`cfront`** 3.0 mode. With this feature enabled, the following code compiles without error:

```
struct A {
  A();
  operator int *();
  int operator[](unsigned);
};
void main() {
  A a;
  a[0];  // Ambiguous, but allowed
         // with this option
         // operator[] is chosen
}
```

As of July 1996, the above is again acceptable, if `ptrdiff_t` is configured as `long`. Using a parameter of type `ptrdiff_t` (instead of `unsigned int`) is recommended for portability.

`--alternative_tokens`
`--no_alternative_tokens`

Enable or disable recognition of alternative tokens. This controls recognition of the digraph tokens in C and C++, and controls recognition of the operator keywords (e.g., `and`, `bitand`, etc.) in C++. The default is **`--alternative_tokens`**.

`--multibyte_chars`
`--no_multibyte_chars`

Enable or disable processing for multibyte character sequences in comments, string literals, and character constants. Multibyte encodings are used for character sets like the Japanese SJIS.

The default is **`--no_multibyte_chars`**.

`--inlining`
`--no_inlining`

Enable or disable function inlining. If disabled, calls to inline functions will call out-of-line instances.

The default is **`--inlining`**.

`--pack_alignment` *n*

Set the default alignment for packing classes and structs to *n*, a power-of-2 value. The argument *n* is the default maximum alignment for nonstatic data members; it can be overridden by a `#pragma pack` directive.

`-Xt`
`--svr4`
`--no_svr4`

Enable or disable recognition of SVR4 C compatibility features. This option also specifies that the source language being compiled is ANSI C.

The default is **`--no_svr4`**.

`--nonconst_ref_anachronism`
`--no_nonconst_ref_anachronism`

Enable or disable the anachronism of allowing a reference to `nonconst` to bind to a class rvalue of the right type. This anachronism is also enabled by the **`--anachronisms`** option and the **`cfront`**-compatibility options.

`--embedded_c++`

Enable the diagnosis of noncompliance with the “Embedded C++” subset (from which templates, exceptions, namespaces, new-style casts, RTTI, multiple inheritance, virtual base classes, and mutable are excluded).

`--enum_overloading`
`--no_enum_overloading`

Enable or disable support for using operator functions to overload built-in operations on `enum`-typed operands.

`--const_string_literals`
`--no_const_string_literals`

Control whether C++ string literals and wide string literals are `const` (as required by the standard) or non-`const` (as was true in earlier versions of the C++ language).

`--class_name_injection`
`--no_class_name_injection`

In C++, controls whether the name of a class is injected into the scope of the class (as required by the standard) or is not injected (as was true in earlier versions of the C++ language).

`--arg_dep_lookup`
`--no_arg_dep_lookup`

In C++, controls whether argument dependent lookup of unqualified function names is performed.

`--friend_injection`
`--no_friend_injection`

In C++, controls whether the name of a class or function that is declared only in `friend` declarations is visible when using the normal lookup mechanisms. When friend names are injected, they are visible to such lookups. When friend names are not injected (as required by the standard), function names are visible only when using argument-dependent lookup, and class names are never visible.

`--late_tiebreaker`
`--early_tiebreaker`

Select the way that tie-breakers (e.g., cv-qualifier differences) apply in overload resolution. In "early" tie-breaker processing, the tie-breakers are considered at the same time as other measures of the goodness of the match of an argument value and the corresponding parameter type (this is the standard approach). In "late" tiebreaker processing, tie-breakers are ignored during the initial comparison, and considered only if two functions are otherwise equally good on all arguments; the tie-breakers can then be used to choose one function over another.

`--nonstd_using_decl`
`--no_nonstd_using_decl`

In C++, controls whether a nonmember using-declaration that specifies an unqualified name is allowed.

--designators
--no_designators

Enable or disable support for designators (a C9X extension). These options are not available in C++ mode.

--extended_designators
--no_extended_designators

Enable or disable support for "extended designators," an extension accepted only in C mode to emulate the behavior of certain other C compilers when it comes to designators in aggregate initializers.

--variadic_macros
--no_variadic_macros

Enable or disable support for variadic macros (a C9X extension that is also available in C++ mode).

--extended_variadic_macros
--no_extended_variadic_macros

Enable or disable support for "extended variadic macros," an extension that emulates the behavior of certain other C compilers when it comes to variadic macros.

--compound_literals
--no_compound_literals

Enable or disable support for "compound literals" (a C9X extension).

--base_assign_op_is_default
--no_base_assign_op_is_default

Enable or disable the anachronism of accepting a copy assignment operator that has an input parameter that is a reference to a base class as a default copy assignment operator for the derived class. This option is enabled by default in **cfront** compatibility mode.

--float_single
-fsingle

Cause all floating-point constants to have type `float` instead of the default type `double`. This can be used to prevent type promotion of floating-point expressions involving constants to double precision.

Note that it is possible to force individual floating point constants to have type float by adding an `f` or `F` suffix. For example, `3.13f` has type `float` while `3.14` has type `double` (by default). This eliminates the need to use the **-fsingle** option and allows greater flexibility in controlling the types of floating-point literals.

--float_single2
-fsingle2

Like the **-fsingle** option, but also disable the automatic type promotion of floating-point expressions to type `double` when passed as parameters to functions. With this option, it is possible to write and use true single-precision functions, but it becomes the user's responsibility to provide double-precision arguments to functions that expect them (such as standard library routines like `printf`).

Note that function prototypes may be used to declare routines that accept float arguments, regardless of default type-promotion rules. This can eliminate the need to use the **`-fsingle2`** option. In ANSI mode, prototype declarations for the single-precision math library routines are available in `<math.h>`. See **`trig(3M)`**.

`--float_mode=`*floatmode*
`-f`*floatmode*

Use *floatmode* as the floating-point mode during compilation and as the floating-point mode of the resulting object file.

floatmode can be any of the following:

`IEEE-COMPATIBLE`

Use mode **`ieeecom`**

`IEEE-ZERO`

Use mode **`ieeezero`** or **`zero`**

`IEEE-NEAREST`

Use mode **`ieeenear`**, **`near`**, or **`ieee`**; this is the default.

`IEEE-POS-INFINITY`

Use mode **`ieeepos`** or **`pos`**

`IEEE-NEG-INFINITY`

Use mode **`ieeeneg`** or **`neg`**

`--enable_intrinsics`
`-F`

Turn on intrinsic functions. The compiler will then generate inline code for accessing special machine instructions. Use of this option also defines the preprocessor macro `_FAST_MATH_INTRINSICS`.

`--read_only_literals`
`-R`

Make initialized variables shared and read-only.

`--namespace_in_headers`
`--no_namespace_in_headers`

Define entities mandated by the ANSI/ISO C standard in the namespace std or in the global namespace. Entities are defined in namespace `std` by default when namespaces are enabled.

`--ansi_cplusplus_headers`
`--no_ansi_cplusplus_headers`

Enable or disable the overloaded declarations of standard C functions which are mandated by the C++ standard. The overloaded declarations are enabled by default.

`--const_object_in_nonconst_member`

Enable the anachronism of calling a member function that does not require a `const` this pointer with a `const` selector. This anachronism is also enabled by the **`--anachronisms`** option and the **`cfront`**-compatibility options.

`--check_long_long`
`-Qcheck_long_long`

Define the predefined macro `__LONG_LONG` to condition header files to allow long long types and signatures. In order to get warning messages whenever these types are seen supply the command line option **`--strict_warnings`**.

`--long_long`
`--no_long_long`
`-Qlong_long`
`-Qno_long_long`

Enable or disable the extension `long long`, a 64-bit integer. Also defines the predefined macro `__LONG_LONG`.

`--const_constant_is_constant`
`-Qconst_constant_is_constant`

In C mode, when a `const` variable is initialized with a constant literal, replace references to the variable with the constant literal. This is always done in C++.

`-K`

Treat `const`, `signed`, and `volatile` as normal identifiers, not as ANSI-C keywords.

`--vectors`
`--no_vectors`

Enable or disable AltiVec vector enhancements (`vector` and `pixel` keywords; see also **`--bool`** to enable the `bool` keyword). Valid only if the target (see **`--target`** option) is one which supports the AltiVec instruction set. Refer to the Motorola document *AltiVec Technology Programming Interface Manual* for details.

The default is always **`--no_vectors`**.

Optimization

The following are **`ec[++]`** *options* related to optimization:

-O[*keyword***[,***keyword2***...]]**

Control the level of optimization performed during compilation. Previous choices are preserved if not overridden, thus multiple **-O** options may be used. Valid keywords are:

--optimization_level=none
--optimization_level=0
-Onone or -O0

Places strict controls on minimal optimization. Usually used only on extremely large, usually machine generated, source files.

--optimization_level=minimal
--optimization_level=1
-Ominimal or -O1

Perform some minimal level of optimization that will yield reasonably fast code. This is the default if no **-O** option is specified.

--optimization_level=global
--optimization_level=2
-Oglobal or -O2

Perform some routine-wide optimizations. This is the default if **-O** is specified with no explicit level.

--optimization_level=maximal
--optimization_level=3
-Omaximal or -O3

Perform all routine-wide optimizations, but with restraint to limit compile time.

--optimization_level=ultimate
--optimization_level=4
-Qbenchmark
-Oultimate or -O4

Perform maximal optimizations, uses fastest transformations, link with fastest libraries, and take lots of time to do it.

--safe
-Osafe

Set the optimization class to **safe**. Avoids all transformations that might cause subtle differences in program behaviour. See also **-Qopt_class**.

--standard
-Ostandard

Set the optimization class to **standard**. Allows transformations that might cause differences in program behaviour if the language standard permits them. See also **-Qopt_class**.

--unsafe
-Ounsafe

Set the optimization class to **unsafe**. See also **-Qopt_class**.

--reorder
-Oreorder

Perform instruction scheduling (default for global and higher)

--no_reorder
-Onoreorder
-Ono_reorder

Do not perform instruction scheduling (default for minimal and lower)

--post_linker
-Oanalyze
-Opost_linker

Perform post-linker optimization (default for global and higher).

--no_post_linker
-Ono_analyze
-Ono_post_linker

Do not perform post-linker optimization (default for minimal and lower)

-Q*flag*

Provide access to a number of special-purpose compiler control options. Many of these options specify optimization parameters that are common to all Concurrent CCG-based compilers; see the "Program Optimization" chapter of Compilation Systems Volume 2 (Concepts) for a detailed description of CCG optimizations and the use of these options. The following flags are available:

--alias_array_elements_limit=*N*
-Qalias_array_elements_limit=*N*

Limit the number of constant array references that are given separate object numbers. For arrays of structures this controls the number of structures to be assigned object numbers. A value of zero means no limits will be applied. Used when **-Qprecise_alias** is in effect. Default value is 100.

--alias_ignore_const
-Qalias_ignore_const

Instruct the alias analysis to ignore the const keyword in performing alias analysis. By default variables that have this attribute are assumed to be constant.

--alias_object_limit=*N*
-Qalias_object_limit=*N*

Limit the number of objects used in alias analysis, which is a measure of the preciseness of the analysis. A value of zero mans no limits will be applied. Used when **-Qprecise_alias** is in effect. Default value is 10000.

--alias_structure_fields_limit=*N*
-Qalias_structure_fields_limit=*N*

Limit the number of fields in a structure that are given separate object numbers. A value of zero means that no limit will be applied. Used with **-Qprecise_alias** is in effect. Default value is 100.

--complete_unroll_debugging
-Qcomplete_unroll_debugging

Generate complete debugging information for basic blocks duplicated during loop unrolling and for zero trip loop optimization at the cost of increased compilation time and object module size.

--dont_peel_var
-Qdont_peel_var

Specifie that loop unrolling should not peel any iterations from loops with unknown iteration count. This is the default since peeling iterations from such loops can adversely effect caching in loops that only execute a few times.

--flttrap
-Qflttrap

Enable software floating-point exceptions. This also disables hardware floating-point exceptions. See **-Qfpexcept**.

--fpexcept={precise | imprecise | disabled}
-Qfpexcept={precise | imprecise | disabled}

Control hardware floating-point exception handling. This option is passed to the linker. See **ld(1)**.

--full_debug_line_info
-Qfull_debug_line_info

Produce debug information when inlining and other optimizations that copy and move code so that the debugger is aware that such a transformation has happened. This is the default. See also **-Qsparse_debug_line_info**.

--growth_limit=*N*
-Qgrowth_limit=*N*

Limit the amount of intermediate code the optimizer is allowed to duplicate when performing optimizations such as loop unrolling and repairing irreducible flow graphs. The integer constant *N* represents the percentage increase in code size permitted. The default value is 25.

--huge_heuristic=*N*
-Qhuge_heuristic=*N*

Limit the aggressiveness of register allocation. Values of *N* may be from 1 through 1000000. The default is 1000000. Very low values (ie, below 100) are most likely to significantly reduce the time required to optimize very large, complicated subprograms using many variables at the cost of reduced efficiency of register usage.

--inline="*list***"**
-Qinline="*list***"**

Inline the comma-separated *list* of routine names. These routines cannot be class names, cannot be overloaded, and cannot be in namespaces.

--inline_depth=*N*
-Qinline_depth=*N*

Place a limit on the level of nested procedure inlining. A value of zero disables inlining. By default it is set to one at minimal or less and two at global and above. See also **--no_inlining**. Setting *N* higher than 2 can result in runaway program size growth and exteremly long compile times.

--int_div_exception
-Qint_div_exception

Generate an exception if an integer division by zero is detected at run time. The PowerPC architecture does not provide this facility without software support. Use of this option instructs the compiler to provide this software support at the expense of integer divide performance.

--invert_divides
-Qinvert_divides

Host divides by region constants (an expression whose value will not change during the execution of the loop containing it) out of loops and replace them with a multiply by the reciprocal in the loop. Also, transform divide by a literal into a multiply by its reciprocal.

--loops=*N*
-Qloops=*N*

Set the number of loops for which the compiler will perform the copy-variable optimization. *N* must be an integer constant. The default value is 20.

--no_float_reg_agg_move
-Qno_float_reg_agg_move

Disable the use of floating point registers to accelerate the copying of structures.

--float_varargs
-Qfloat_varargs

--no_float_varargs
-Qno_float_varargs

Tell the compiler that there will be no floating point arguments passed to any stdarg or vararg routine. This enables the compiler to generate faster code by avoiding the spill of floating point registers to a buffer. The default is always **--float_varargs**.

--vector_varargs
-Qvector_varargs
--no_vector_varargs
-Qno_vector_varargs

Tell the compiler that there will be no AltiVec vector-type arguments passed to any stdarg or vararg routine. This enables the compiler to generate faster code by avoiding the spill of vector point registers to a buffer. The default is **--vector_varargs** if the target (see **--target**) supports AltiVec instructions, and **--no_vector_varargs** otherwise.

--vector_safe_prologs
-Qvector_safe_prologs
--no_vector_safe_prologs
-Qno_vector_safe_prologs

Tell the compiler to generate a runtime check around function prolog code to avoid executing AltiVec instructions when code compiled for an AltiVec supporting target is ran on a non-AltiVec supporting processor.

--no_invert_divides
-Qno_invert_divides

Disable the implicit **-Qinvert_divides** present at ultimate optimization.

--no_multiply_add
-Qno_multiply_add

Disable the use of multiply-add instructions. Separate multiply and add instructions will be used instead.

-Qreentrant_library

Disallow the implied use of the non reentrant C library `libnc` at ULTIMATE optimization.

--report_optimizations
-Qreport_optimizations

Report on what optimizations are being performed.

--objects=*N*
-Qobjects=*N*

Set the maximum number of variables that the compiler will optimize when global or maximal optimization is specifed. *N* must be an integer constant. (Limits optimizations such as dead code elimination, copy propagation and copy variables). The default is 128.

-Qopt_class={unsafe | standard | safe }

Select the class of compiler optimization. The default is **unsafe**. See also **--unsafe**, **-Ounsafe**, **--standard**, **-Ostandard**, **--safe**, and **-Osafe**.

--optimize_for_space
-Qoptimize_for_space

Make space rather than time the critical factor in optimizing this program.

--peel_limit_const=*N*
-Qpeel_limit_const=*N*

Specify the minimum number of iterations that loop unrolling will peel from a loop whose number of iterations is a compile-time constant. This is used for software pipelining so that each iteration of a loop can overlap instructions from *N*+1 iterations of the original loop. The default is 1 at global and 2 at maximal and up.

--peel_var
-Qpeel_var

Specify that loop unrolling can pull 1 iteration off a loop with an unknown iteration count. The default is **-Qdont_peel_var**.

--plt
-Qplt

Generate function calls using plt instead of gotp entries if **-Zpic** also specified. See Chapter 22 of *Compilation Systems Volume 2 (Concepts)* for the cases when it is desirable to use this option.

--precise_alias
-Qprecise_alias

Specify that the time consuming, but precise, method of alias analysis is used. It is the default at global and above.

--quick_alias
-Qquick_alias

Specify that the quick, but not very precise, method of alias analysis is used. It is the default at minimal and below.

--sparse_debug_line_info
-Qsparse_debug_line_info

Produce smaller debug info for inlining and other optimizations that copy and move code. Debugging inlined procedures, unrolled loops, etc. will be somewhat more difficult. See also **-Qfull_debug_line_info**.

--spill_register_if_address_taken
-Qspill_register_if_address_taken

Copy the contents of all argument register to memory locations if the address of a formal parameter is taken. This option can sometimes be used to work around problems with functions that attempt to step through argument lists assuming that they were passed in as a list of arguments on the stack. Typically, these functions should have been coded using the `varargs(5)` interface for passing variable argument lists to functions.

--target={ppc604|ppc604e|ppc750|ppc7400|ppc}
-Qtarget={ppc604|ppc604e|ppc750|ppc7400|ppc}

Select the target architecture family for compilation and optimization. The default is **ppc750**. The more general target **ppc** produces code using only instructions common to the entire PowerPC CPU family.

The target may also be specified by setting the `TARGET_ARCH` environment variable.

--unroll_limit_const=*N*
-Qunroll_limit_const=*N*

Limit the number of times a loop with a number of iterations known at compile time may be unrolled. For more information, see the “Program Optimization” chapter of the *Compilation Systems Volume 2 (Concepts)*. *N* must be an integer greater than or equal to 0. The default is 10.

--unroll_limit_var=*N*
-Qunroll_limit_var=*N*

Limit the number of times a loop with an unknown number of iterations may be unrolled. See also **-Qunroll_limit_const**.

Linking

The following are **ec[++]** *options* related to linking:

--dynamic_link_name=*name*
-*hname*

Passed to **ld(1)** in dynamic mode only. When building a shared object, put name in the object’s dynamic section. name is recorded in executables that are linked with this object rather than the object’s system file name. Accordingly, name is used by the dynamic linker as the name of the shared object to search for at run time. (See **-l** and **-o**.)

--library=*name*
-l*name*

Passed to **ld(1)**, to search for the library **libname.a** from **/usr/ccs/lib** or **/usr/lib**, and load referenced modules from that library into the executable file. Multiple **-l** options may be used to specify multiple libraries to search. (See **-h** and **-o**).

-lnc

Link with the nonreentrant C library **libnc**, which has better performance compared with **libc** while sacrificing use with threaded or dynamically-linked executables. This option cannot be used when linking dynamically or with the **-lthread** library This option is implied by the **-O4** option when linking a static executable, but can be disallowed by specifying the **-Qreentrant_library** option.

--library_directory=*path*
-L*path*

Passed to **ld(1)**, to add path to the library search directories. **ld** searches for libraries first in any directories specifeid with **-L** options, then in the standard directories. This option is effective only if it precedes the **-l** option on the command line. Multiple **-L** options may be used to specify multple paths to search.

-Z*flag*

Provided access to a number of options that control the object file formats used by C/C++. See *Compilation Systems Volume 1 (Tools)* and *Compilation Systems Volume 2 (Concepts)* for more information about the object-file formats and the tools that deal with them. (Also see the **-h** option). The following flags are available:

--library_linkage={dynamic | static}
-Zlibs={dynamic | static}

Govern library inclusion. **-Zlibs=dynamic** is valid only in dynamic mode (see **-Zlink=static**) in which case it is the default value. If the **-Zlibs=static** option is given, no shared objects will be accepted until **-Zlibs=dynamic** is seen.

--link_mode={dynamic | static}
-Zlink={dynamic | static}

Produce dynamically-linked or statically-linked object files. Dynamic linking is the default unless the environment variable `STATIC_LINK` is defined.

--link_mode=so
-Zlink=so

Produce a shared object instead of an executable. Requires that all objects be compiled with the **-Zpic** option.

--pic
-pic
-Zpic

Cause the compiler to produce position-independent code. Use in conjunction with **-Zlink=so**.

--symbolic
-Zsymbolic

In dynamic mode only, when building a shared object, bind references to global symbols to their definitions within the object, if definitions are available. Normally, references to global symbols within shared objects are not bound until run time, even if definitions are available, so that definitions of the same symbol in an executable or other shared objects can override the object's own definition. **`ld(1)`** issues warnings for undefined symbols unless **`ld`**'s **`-zdefs`** overrides.

`--combine_relocatable_objects`
`-r`

Combine relocatable object files to produce one relocatable object file. **`ld(1)`** will not complain about unresolved references. This option cannot be used in dynamic mode.

`-Qno_vendor_reloc`

Do not generate relocation in the vendor section. This disables the ability of **`analyze(1)`** to optimize a program.

`--entry_point=`*symbol*

Set the entry point address for the executable to be the address fo *symbol*.

`--linker_z=`*flag*
`-z`*flag*

Pass **`-z`** option to **`ld(1)`**.

`--mapfile=`*filename*
`-M`*filename*

Read filename as a text file of directives to **`ld(1)`**. Because these directives change the shape of the output file created by **`ld`**, use of this option is strongly discouraged.

`--memory_map`
`-m`

Produce a memory map or listing of the input/output sections on **`stdout`**. See **`ld(1)`**.

`--reduced_symbols`
`-x`

Do not preserve local symbols with type `STT_NOTYPE`. This option saves some space in the output file.

`--strip`
`-s`

Strip symbolic information from the output file. The debug and line sections and their associated relocation entries will be removed. Except for relocatable files or shared objects, the symbol table and string table sections will also be removed from the output object file.

`--undefined_linker_symbol=`*symbol*
`-u`*symbol*

Enter *symbol* as an undefined symbol in the symbol table. This is useful for loading entirely from an archive library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. The placement of this option on the command line is significant; it must be placed before the library that will define the symbol.

`--no_demangling`

Disable demangling symbol names in error messages coming from /bin/ld when linking in C++ mode.

`--prelink_objects`

Run the prelinker on a set of objects files to resolve instantiatable entities, but don't link them together into an executable or shared object.

`--static_Cruntime`

When generating a shared object and linking in C++ mode, force the implicit inclusion of the C++ runtime to be statically linked rather than dynamically linked.

2
Using the Program Development Environment

2
Using the Program Development Environment

The Concurrent C/C++ Program Development Environment consists of a number of utilities that provide support for library management, compilation, program generation, and debugging.

This chapter provides an simple tutorial which will guide the user in using a number of these utilities. See Chapter 4 (“Program Development Environment Utilities”) for more detailed information on each of these utilities.

Hello World - An Example

To demonstrate the ease of use of Concurrent C/C++ Program Development Environment, a simple example will be given. This example will traverse through the core functions needed to build an executable under the Concurrent C/C++ system.

Building an executable under Concurrent C/C++ can be broken down into as few as four steps:

- Creating an environment
- Introducing units
- Defining a partition
- Building the partition

This section will demonstrate each of these steps on a simple, but well-known example - Hello World.

Before we begin...

You must make sure that the path **`/usr/ccs/bin`** is added to your `PATH` environment variable. This is the only path necessary to access the Concurrent C/C++ utilities, regardless of the number of releases of Concurrent C/C++ installed on the system.

Creating an environment

One of the first steps you must take in order to use Concurrent C/C++ is to create an *environment*. Concurrent C/C++ uses environments as its basic structure of organization. Environments contain all the information relevant to a particular project. All of the Concurrent C/C++ utilities work within the context of a particular environment.

The Concurrent C/C++ tool used to create an environment is **`c.mkenv`**. It requires a Unix directory where this environment will reside.

For our example, we will create a new directory on our system and run **`c.mkenv`** from within that directory.

```
$ mkdir /pathname/earth
$ cd /pathname/earth
$ c.mkenv
```

Screen 2-1. Creating an environment

This creates the Concurrent C/C++ internal directory structure that comprises the environment and that is essential before any other Concurrent C/C++ tools can be utilized. This environment has the same name as the directory in which it was created. Our environment in this example, therefore, is **`/pathname/earth`**.

Introducing units

Compilation units (henceforth referred to simply as *units*) are the basic building blocks of Concurrent C/C++ environments. It is through units that Concurrent C/C++ performs most of its library management and compilation activities. These units are, however, introduced into the system in the form of *source files*.

In our example, we have one unit, `hello`, that resides in a source file, **`hello.C`**. This source file is just an ordinary text file. By default, the name of the unit is the file name of the source file without any path prepended or extension postpended.

The source file, **`hello.C`**, is shown below:

```
#include <iostream>
main() {
   std::cout << “Hello World!!!\n”;
}
```

Create this source file within the directory in which you created your environment. (It is not necessary for the source file to reside in the same directory as the environment. You may specify a relative or absolute path name of the source file.)

We introduce this unit to the environment by using the **`c.intro`** utility. (See “Invoking the Compiler” on page 1-13) **`c.intro`** introduces a unit for source file into the current environment.

```
$ c.intro hello.C
```

Screen 2-2. Introducing a unit from a source file

The unit `hello` is now a part of the environment **`earth`**.

From this point on, the unit `hello` is considered to be *owned* by the environment **`earth`**. Any functions performed on this unit must be managed by the environment through the Concurrent C/C++ utilities. See Chapter 4 (“Program Development Environment Utilities”) for more detailed information on each of these utilities.

Defining a partition

If we want to create an executable program to use our unit, we must define a *partition*. We will be creating an *executable partition* which is the type that corresponds to executable programs.

We must also name the partition. You can name your partition anything you want and then add units to it, but since this is a simple example, we are taking the most direct route.

Hence, our partition will be named **`hello`**. We wUill use the Concurrent C/C++ utility **`c.partition`** to do this.

```
$  c.partition -create executable hello
```

Screen 2-3. Defining a partition

Because it has the same name as the executable partition being created, the unit `hello` is automatically added to this partition.

NOTE

The command in Screen 2-3 could have been explicitly specified as:

```
$  c.partition -create executable -add hello hello
```

This command creates an active partition named **`hello`** and adds the `hello` unit to it.

Building a partition

The last step now is to build the executable. All the necessary steps have been done. Just issue **`c.build`**. This will build an executable file that you can run.

```
$  c.build
```

Screen 2-4. Building a partition

Because no arguments were specified, **`c.build`** tries to build everything it can within this environment. Since we've only defined one unit, `hello`, contained in one partition, **`hello`**, it will only build that.

Success!!!

Now all that's left is to run the program as you would any other executable program. Enter the name of the executable, in this case **`hello`**.

```
$  ./hello
Hello World!!!
$
```

Screen 2-5. Executing the program

And there you have it! Your program has successfully been built and run.

Let's look around...

Now that we have some substance to our environment, let's take a look around and see what things look like. We can use some of the Concurrent C/C++ utilities to investigate the state of our environment and what's in it.

Listing the contents of your environment

Something you might want to do is to see what units and partitions are contained within this environment. **`c.ls`** provides this list for you. **`c.ls`** provides many different options, allowing you to sort the list by some attribute or filter the units based on certain criteria. We'll just take a look at a basic list of the contents of the environment. This is done by issuing the **`c.ls`** command with no options from within your current environment.

```
$ c.ls

PROJECT /pathname/earth
    frozen                    : no
    language                  : C++

    units:
        hello

    partitions:
        hello

$
```

Screen 2-6. Listing the units in an environment

You may want to see more information. You can do this by specifying the **`-1`** or **`-l`** options to the **`c.ls`** command which will give you a long listing. (Even more information can be seen by specifying the **`-v`** option.)

```
$ c.ls -1

PROJECT /pathname/earth
   frozen                    : no
   language                  : C++

   UNIT            LANGUAGE        SOURCE FILE      EFFECTIVE OPTIONS
    hello           C++             hello.C

   PARTITION       KIND            PATH
    hello           executable      hello
$
```

Screen 2-7. Listing the units in an environment (`-l` option)

Viewing the source for a particular unit

Once you know what units are in your environment, you may want to see the source for a particular unit. The Concurrent C/C++ utility **`c.cat`** outputs the source of a given program unit. It outputs a filename header for the source file by default, but this can be suppressed by specifying the option **`-h`**.

The following figure shows how to view the source for the unit `hello` using **`c.cat`**.

```
$ c.cat hello
********** hello.C **********
 #include <iostream>
 main() {
    std::cout << "Hello World!!!\n";
 }
$
```

Screen 2-8. Viewing the source for a particular unit

Looking at the Environment Search Path

Each Concurrent C/C++ environment has an Environment Search Path associated with it. The Environment Search Path is your gateway to other environments. You can list your Environment Search Path by using the **c.path** utility.

```
$  c.path  -v
Environment Search Path:
$
```

Screen 2-9. Viewing your Environment Search Path

Using the Environment Search Path, you can use units that exist in foreign environments. All you need to do is add the environment's path to your Environment Search Path. It's as simple as that!

What are my options?

Concurrent C/C++ uses the concept of persistent compile options. These options are specified through **c.options** and are "remembered" at compilation time. They can apply to any of four areas: permanent environment-wide compile options (which apply to all units within the environment), temporary environment-wide compile options (which temporarily override the permanent ones), permanent unit options and temporary unit options (both of which apply and are unique to specific units).

Let's manipulate the options in our example to give an idea of how it all works.

First, we will consider the environment-wide compile options. These apply to all the units within the environment. Since we only have one unit right now, it will apply to that. However, if we add any others later, they will "inherit" these options automatically.

The *environment-wide compile options* are referenced by the **-default** flag to **c.options**. We'll use the **-list** flag to display what they're set to now:

```
$ c.options -list -default
permanent options:
temporary options:
effective options:
```

Screen 2-10. Listing the environment-wide compile options

You'll see that nothing is listed. That's because we haven't set anything yet. So let's set them to something and see what happens.

`c.options` provides the **`-set`** option to initialize or reset an option group. Let's set our environment-wide compile option set to contain the options **`-g`** and **`-O2`**. (These turn on the generation of debug information and set the optimization level to `GLOBAL`, respectively. You can find out all about these options in "Invoking the Compiler" on page 1-13)

```
$  c.options -set -default -- -g -O2
```

Screen 2-11. Setting the environment-wide compile options

The "**`--`**" sets off the options to **`c.options`** itself from the options for the compiler. Now let's list them again to see if they've taken effect:

```
$ c.options -list -default
permanent options: -g -O2
temporary options:
effective options: -g -O2
$
```

Screen 2-12. Listing the environment-wide compile options (after `-set`)

We can see that the environment-wide compile option set now consists of **`-O2`** and **`-g`**. The effective options line shows what options are effective after the temporary options have overridden some or all of the permanent options.

Remember, these options apply to all units in the environment and will be "inherited" by any units we add to this environment.

If we'd like to set particular options for a specific unit, we can use the *permanent unit compile options* for that unit. They're set in much the same way as environment-wide options, except that we need to specify the units to which they apply.

Let's set the permanent options for the unit `hello` so it is compiled at a `MAXIMAL` optimization level (**`-O3`**). This is done with the following command:

```
$ c.options -set -- -O3 hello
$ c.options -list all
UNIT hello
   permanent options: -O3
   temporary options:
   effective options: -g -O3
$
```

Screen 2-13. Setting the permanent unit options for `hello` unit

We may decide that in addition to the specified options, we may want to "try out" some options or change particular options for a specific compilation but only "temporarily". The *temporary environment-wide default and unit compile options* are for this purpose.

Say we want to produce no debug information for our `hello` unit for this particular compilation. We can set a temporary compile option for that.

```
$  c.options -set -temp -- -!g hello
```

Screen 2-14. Setting the temporary unit options for `hello` unit

In addition, we remember that we also want to limit the depth that function inlining is done. We can "add" this to the temporary option set by using the **`-mod`** flag to **`c.options`**.

```
$  c.options -mod -temp -- -Qinline_depth=1 hello
```

Screen 2-15. Modifying the temporary unit options for `hello` unit

If we list the temporary options for the unit `hello`, we will see that we now have **`-!g`** and **`-Qinline_depth=1`** in the temporary option set:

```
$ c.options -list -temp hello
UNIT hello
   temporary options: -!g -Qinline_depth=1
$
```

Screen 2-16. Listing the temporary options for `hello` unit

These four option sets have a hierarchical relationship to one another which means that the permanent environment-wide compile options are overridden by the temporary environment-wide options which are, in turn, overridden by the permanent unit options which are, in turn, overridden by the temporary unit options. This relationship forms the *effective compile options* for the unit, which the compiler will use during compilation. We can see these in Table 2-1:

Table 2-1. Effective options for `hello` unit

Permanent environment-wide options	**-g**	**-O2**	
Temporary environment-wide options			
Permanent unit options		**-O3**	
Temporary unit options	**-!g**		**-Qinline_depth=1**
EFFECTIVE OPTIONS		**-O3**	**-Qinline_depth=1**

If we list the effective options for the `hello` unit, we will see similar results:

```
$ c.options -eff hello
UNIT: hello
   effective options: -O3 -Qinline_depth=1
$
```

Screen 2-17. Listing the effective options for `hello` unit

If, after we compile with these options, we find any particular option that we would like to delete, we can do so by using the **`-del`** flag. For example, let's delete the inline depth option from the temporary options.

```
$  c.options -del -temp -- -Qinline_depth=1 hello
```

Screen 2-18. Deleting from the temporary options set for `hello` unit

And if we like the other temporary options so much that we'd like to make them permanent, Concurrent C/C++ provides the **-keeptemp** flag to propagate all the temporary options for a particular unit to the permanent option set for that same unit. If we do this,

```
$  c.options -keeptemp hello
```

Screen 2-19. Propagating the temporary options to the permanent set

the temporary option **-!g** will become a permanent unit option for the unit `hello`.

The effective options will now resemble that of Table 2-2:

Table 2-2. Effective options for `hello` unit (after -keeptemp)

Permanent environment-wide options	**-g**	**-O2**	
Temporary environment-wide options			
Permanent unit options	**-!g**	**-O3**	
Temporary unit options			
EFFECTIVE OPTIONS		**-O3**	

If we list the effective options for the `hello` unit, we will see similar results:

```
$ c.options -list hello
UNIT hello
   permanent options: -!g -O3
   temporary options:
   effective options: -O3
$
```

Screen 2-20. Listing the effective options for `hello` unit (after -keeptemp)

See "c.options" on page 4-49for a complete description of the functionality of this Concurrent C/C++ utility.

Hello Galaxy - The Example Continues...

Let’s set up another environment with a function that our `hello` unit can contact.

Let’s set up a new environment, **`galaxy`**, and introduce a source file very similar to **`hello.C`**. We’ll call this file **`alien.C`** and it will contain the following unit, `alien`. The file is shown below:

```
#include <iostream>
void planet() {
   std::cout << “Greetings from Outer Space!!!\n”;
}
```

Create a different directory **`/pathname/galaxy`** to contain our new environment and place the source file, **`alien.C`**, in it. From within that directory, the following commands will create our environment and introduce the source file into it.

```
$  c.mkenv
$  c.intro alien.C
```

Screen 2-21. Setting up another environment

NOTE

We have not compiled this unit nor have we created a partition and included the unit in the partition to be built. This was intentional to demonstrate a point later in the example.

Modifying an existing unit

Now we must go back to our original environment **`earth`** that contains our original unit `hello`.

We will update the unit `hello` so that it references the new `alien` unit. We do this by using the **`c.edit`** utility. **`c.edit`** edits the source file that contains the unit specified. It does this by using the editor referenced in the `EDITOR` environment variable. It then updates the environment so that the automatic compilation utility, **`c.build`**, knows that this unit needs to be rebuilt.

NOTE

`c.edit` is the supported method for modifying units that have been introduced into the environment. Any modifications to the units other than through the tools provided is discouraged, although the tools support it.

Specify the unit name to the **`c.edit`** command.

```
$  c.edit hello
```

Screen 2-22. Editing a unit

Add the following lines (indicated in **bold**) to the source file containing the unit `hello`:

```
#include <iostream>
extern void planet();
void planet() {
   std::cout << “Greetings from Outer Space!!!\n”;
   planet();
}
```

Save the changes to the file.

Building a unit with references outside the local environment

Now let’s try to build it.

Issue the **`c.build`** command as before.

```
$ c.build
Undefined                           first referenced
 symbol                                  in file
planet()                           .c++/.units/hello.o
ld: hello: fatal error: Symbol referencing errors. No output written to hello
ec++: ERROR: Errors in the ld pass, status = 1
c.build: failed building partition hello
c.build: there was a failure building one or more partitions
$
```

Screen 2-23. Building the partition with reference to `alien` unit

Because the `alien` unit does not exist in the current environment AND because we have not manually added it to our Environment Search Path, **`c.build`** cannot find it and therefore complains.

Adding an environment to the Environment Search Path

This is easily remedied by adding the new environment's path to the Environment Search Path for the **`earth`** environment using the **`c.path`** utility.

You can see that it has been added to your Environment Search Path by issuing the **`c.path`** command with no parameters again.

```
$ c.path -A /pathname/galaxy
$ c.partition -add alien hello
$ c.path -v
Environment Search Path:
   /pathname/galaxy
$
```

Screen 2-24. Adding to and viewing the updated Environment Search Path

Making contact!!!

Now try to issue **`c.build`** again. This time it will be successful.

After it is successfully built, run the **`hello`** executable again.

```
$ c.build
$ ./hello
Hello World!!!
Greetings from Outer Space!!!
$
```

Screen 2-25. Executing the new `hello` - contact is made!

Who resides here now?

Let's take a look at who inhabits our environment **`earth`** now. Remember before when we issued the **`c.ls`** command, we saw that our environment contained the lone unit `hello`. Let's issue the command again and see what has happened since we made contact with the `alien.`

```
$ c.ls -all

PROJECT /pathname/earth
   frozen                    : no
   language                  : C++
   default permanent options : -g -O2
   foreign environment path:
      /pathname/galaxy

   units:
      alien
      hello

   partitions:
      hello
$
```

Screen 2-26. Listing the units

You can now see that the unit `alien` has been added to the list of units in this environment. (Foreign units are not listed unless the **`-all`** option is specified).

Although they are both listed *local* to this environment, they each have a different means of citizenship.

- The unit `hello` was introduced directly into this environment. Therefore, it is regarded as a *native* unit.
- The `alien` unit, however, was never formally introduced into the local environment. It was found on the Environment Search Path.

  Now, remember that the `alien` unit was not compiled in its original foreign environment. The **`c.build`** command, when run in this local environment, could not find a compiled form of the `alien` unit on the Environment Search Path and had to do something in order to build the partition. It therefore compiled the `alien` unit in the local environment.

  This compiled form of a foreign unit within the local environment is considered *naturalized* by the system.

NOTE

If the `alien` unit had been compiled in its own foreign environment, **`c.build`** would have found that compiled form on the Environment Search Path and would have used that when linking the **`hello`** executable together.

FURTHER NOTE

The **`-noimport`** option will inhibit the automatic naturalization behavior of **`c.build`**. If it had been used in this example, **`c.build`** would have reported an error.

Conclusion

This concludes the Concurrent C/C++ Program Development Environment tutorial. You should now be familiar enough with the Concurrent C/C++ PDE to explore the additional functionality not covered in this tutorial. You may investigate the other sections of this manual to get more detailed explanations of PDE functionality.

If any questions arise while you are using the Program Development Environment, you may use invoke the online help system by issuing the command:

```
c.man
```

and selecting the Concurrent C/C++ Reference Manual from the bookshelf. See “c.man” on page 4-45 for details.

3
Program Development Environment Concepts

3
Program Development Environment Concepts

Concurrent C/C++ uses the concept of *environments* as its basic structure of organization. These environments take advantage of various utilities provided by Concurrent C/C++ to manipulate *compilation units* (referred to simply as *units*) that may form *partitions*.

Utilities for library management, compilation and program generation, and debugging are provided by Concurrent C/C++.

This chapter will discuss in further detail the concepts of environments, units and partitions and their relationship to library management, program generation, and debugging.

Environments

Concurrent C/C++ uses the concept of environments as its basic structure of organization. Environments may include:

- units that have been introduced
- partitions that have been defined
- Environment Search Paths
- references to source files (which generally contain units)
- other information used internally by Concurrent C/C++

Environments collect and maintain *separate compilation information* which is information collected from previous compilations.

Concurrent C/C++ permits local environments to reference other *foreign* environments thus providing visibility to the units and partitions therein. This feature allows programmers to work on local versions of individual program units while retrieving the remainder of the program from previously developed environments.

A Concurrent C/C++ environment may be initialized or created in any desired location in a filesystem using the **`c.mkenv`** utility.

Concurrent C/C++ provides several other utilities to maintain, modify and report on the contents of environments. Any modifications to the environment other than through the tools provided by Concurrent C/C++ is discouraged, although the tools support it as well as possible.

Local Environments

By default, Concurrent C/C++ uses the current working directory as its *local environment*. All Concurrent C/C++ utilities perform their actions within this local environment unless the **`-env`** option is explicitly specified.

For example, if no environment is specified with the **`c.mkenv`** tool, Concurrent C/C++ will set up its internal directory structure for that environment within the current working directory.

When used with any of the Concurrent C/C++ utilities, however, the **`-env`** option allows the user to specify a target environment other than the current working directory. The actions of the Concurrent C/C++ utility using this option will be performed in the environment specified and not in the local environment. (See Chapter 4 (“Program Development Environment Utilities”) for more details on using this parameter with each of the tools.)

Foreign Environments

Concurrent C/C++ uses the Environment Search Path to reference units within foreign environments. These units can be used as foreign units or can be brought into the local environment through naturalization or fetching.

Environment Search Path

Concurrent C/C++ uses the concept of an *Environment Search Path* to allow users to specify that units from environments other than the current environment should be made available in the current environment. This Environment Search Path relates only to each particular environment and each environment has its own Environment Search Path.

By placing the location of another environment on the *Environment Search Path* for a given environment, all the units from the other environment are conceptually added to the given environment, unless that would involve replacing a unit which was either introduced manually into the environment by a user, or would replace a unit which was introduced from yet a third environment which precedes the other environment in the Environment Search Path. In order to add or delete environments on your Environment Search Path, you may use the **`c.path`** tool. See “c.path” on page 4-57.

In addition to accessing units in foreign environments, the user may also link with partitions (archives, shared-object, and object partitions) that are located in foreign environments. Partition names have the same visibility rules that units do.

Naturalization

At times, it is necessary for the compilation system to make local copies of units that exist in foreign environments. For example, if a foreign unit is referenced within a local unit and no compilation has been done on that foreign unit in that foreign environment, a local copy of the foreign unit will be compiled within the current environment, using any options that would apply to the foreign unit. This happens transparently to the user. Should a naturalized unit subsequently be built within its native environment, then the tools will automatically expel the naturalized copy and begin using the object file in the foreign environment.

Fetching

It may be desirable for users to force copies of specified units from other environments into the current environment. This eliminates any requirement that the unit be compiled in the foreign environment, so long as it is compiled locally. The **`c.fetch`** tool (see "c.fetch" on page 4-24) is provided for that purpose. Units that are fetched also take precedence over units that are in the Environment Search Path. Units may even be fetched from environments that are not on the Environment Search Path.

Freezing Environments

An environment may be frozen using the **`c.freeze`** utility. This changes an environment so that it is unalterable.

A frozen environment is able to provide more information about its contents than one that is not frozen. Therefore, accesses to frozen environments from other environments function much faster than accesses to unfrozen environments.

Any environment which will not be changed for a significant period of time and which will be used by other environments is a good candidate to be frozen to improve compilation performance.

See "c.freeze" on page 4-25 for information on this utility.

Environment-wide Compile Options

Environment-wide compile options apply to all units within an environment. See "Environment-wide Options" on page 3-10.

Units

Compilation units (or simply units) are the basic building blocks of Concurrent C/C++ environments. Instead of dealing with source files for library management and compilation activities, Concurrent C/C++ focuses on the concept of units. A *compilation unit* can be the routines and global data packaged in a primary source file, an extern inline function, or an instantiatable template entity (in C++).

Unit Identification

For many of the Concurrent C/C++ utilities in Chapter 4, the following definition is given:

unit-id is defined by the following syntax:

unit **`| all`**

Nationalities

Compilation units in Concurrent C/C++ have a nationality associated with them. Units can be either *local* or *foreign*.

Local Units

Compilation units that are *local* to a system can be one of three types:

native

Native compilation units are introduced into an environment by using the **`c.intro`** function.

Once a unit is introduced into an environment, it is considered to be owned by that environment and any functions performed on that unit should be managed by the environment through the Concurrent C/C++ utilities.

naturalized

Sometimes, the compiled form of a foreign unit is not available when it is needed locally for a build. In this case, the system automatically makes a local compilation. This local compiled form is considered to be naturalized.

A naturalized unit retains the compile options from its original environment. These options can only be altered by changing them in the original environment.

Naturalized units are automatically expelled from the local environment should an up-to-date version be built in its native environment.

fetched

In some cases, it may be desirable for users to manually fetch unit from another environment into the local environment.

A fetched unit retains the unit-specific options from the original unit but these options may be changed in the local environment. However, it does not retain the environment-wide options of its original environment. It uses those of the current environment instead.

Fetched units must be *expelled* from the environment by using **`c.expel`** if they are no longer desired.

Foreign Units

Foreign units are those units that exist in other environments which are on the Environment Search Path. The user is not required to do anything special in order to use these units. They become automatically available once their environment is added to the Environment Search Path. A foreign unit is marked *visiting* if it is actually used in the construction of a local partition.

Artificial Units

At times, the implementation may create units to fill internal roles such as instantiating template entities or extern inline functions. These units are created, utilized, and sometimes discarded during the compilation phase. The user may use the **`-art`** option to **`c.ls`** to display the artificial units in the environment. See “c.ls” on page 4-36 for more information.

Unit Compile Options

Each unit has a set of permanent and temporary compile options associated with it. These compile options are described in more detail in “Permanent Unit Options” on page 3-10

Partitions

A *partition* is an executable, archive, shared object, or object file that can be invoked or referenced outside of the Concurrent C/C++ Program Development Environment. Partitions consist of one or more units that have been introduced into the environment. The units included in a partition are those that the user explictly assigns and units which they require. Concurrent C/C++ manages these units and their dependencies, as well as link options and configuration information for each partition within the context of an *environment*. A partition definition must include one or more units in order to be built.

A partition within Concurrent C/C++ is created and maintained by using the **`c.partition`** function. This function provides tools to create a partition, add or delete units from a partition, and various other utilities.

In much the same way that options and configuration information concerning compilation are associated with units, linker options and configuration information for linking are associated with partitions. Partitions are basically recipes to the linker which indicate how to build a target file from units.

Types of Partitions

Concurrent C/C++ defines four types of partitions:

- Executable Programs
- Archives
- Shared Objects
- Object Files

Executable Partitions

Executable partitions describe how to build an executable program.

Archives

An *archive* is a collection of routines and data that is associated with an application during the link phase. Archives are useful for linking into other, potentially non-C/C^{++}, applications. Archives are usually designated with a **`.a`** suffix.

Archives differ from shared objects by the form of the object contained within it. Archives contain statically-built (i.e. non-shared) objects within them. (See “Position Independent Code” on page 3-7 for more details)

Shared Objects

A *shared object* is a collection of routines and data that is associated with an application during the link and execution phases. Shared objects are useful for linking into other C/C++ or non-C/C++ applications. Shared objects are usually designated with a **`.so`** suffix.

Shared objects differ from archives by the form of the object contained within it. Shared objects are dynamically built (i.e. shared) objects that contain position independent code. (See “Position Independent Code” on 3-7 for more details)

At link time, routines and data objects from a shared object may satisfy unresolved references from an application, but they are not copied into the resultant application’s executable image. The actual associations and memory allocations occur during the initial phase of the application’s execution; this is termed the *dynamic linking* phase. Because of this, it is possible for shared objects to be changed and these changes to affect the application that has linked with them. However, due to this dynamic linking property of shared objects, it is often not necessary to rebuild the calling application after the shared object has changed.

During dynamic linking, all shared objects that the application requires are allocated and linked into the application’s address space, sharing as many physical memory pages with other concurrently executing applications as possible. Therefore, totally dissimilar applications may share the same physical pages for the same shared object. This applies to the memory for the actual code or machine instructions in the shared object. The memory for the data segments in a shared object is usually replicated for each application using that shared object.

Lazy Versus Immediate Binding

After the dynamic linker successfully locates all of the shared objects required for the application program, it maps their memory segments into the application program's address space.

The dynamic linker uses internal symbol tables to satisfy symbol references in the application program. Entries in these tables describe the final location of symbols found in the shared objects; this is termed *relocation*. All data references are immediately relocated.

By default, the dynamic linker does not fully relocate all subprogram references in the application program (or the shared objects themselves, because they can reference other shared objects or routines in the application program). If an as-yet unrelocated reference occurs, control passes once again to the dynamic linker which then relocates the reference. This is termed *lazy binding*.

To force immediate binding of all references, the user may invoke the program with the `LD_BIND_NOW` environment variable set. See *Compilation Systems Volume 1 (Tools)* for more information.

Position Independent Code

In order to create a shared object, the compiler must generate code in a position-independent manner. *Position independence* refers to the fact that the generated code cannot rely on labels, data, or routines being in known locations; these locations are not known until dynamic linking occurs. *Position independent code* (PIC) requires additional indirections at run-time; therefore, routines within shared objects are inherently slightly slower than non-shared versions of those routines.

You control whether a unit is compiled as position independent code via a compilation option, **`-Zpic`**, set with the **`c.options`** command.

Share Path

Because the actual association of a shared object with a user application does not occur until execution time, the shared object must exist on the target system in a specific location, configurable by the user. By default, the path name of the shared object is that defined by the target of the partition.

When creating a partition, you may specify an alternative path name (or *share path*) for the shared object. The shared object will still be built at the pathname specified for the target, but it must be placed at the share path before any executables using it can be run. This is set by the **`-sp`** link option in the **`c.partition`** command. Alternatively, a soft link can be created by using the **`-sl`** link option in the **`c.partition`** command when defining the shared object.

Issues to consider

While the use of shared objects almost always reduces disk space utilization on the target architecture and often improves development productivity by minimizing application link time, it may or may not actually improve run-time memory utilization. The following issues should be considered.

1. Are the shared objects configured with an appropriate *granularity* (i.e. the number of C/C++ units located in each shared object) with respect to the particular client application programs that will be concurrently executing?

   For example, it is possible that if only two application programs concurrently execute and use large granular shared objects, more memory may potentially be used than in a non-shared object scenario. There is a trade-off between small granularity and manageability.

2. Will the application make use of local memory, and if so, how many applications will be executing out of the same local memory pools using the same shared object?

3. What disk storage capacity does the system have? The difference in size between ordinary objects and PIC objects is negligible. However, if both a shared and static version of a source file is built, then the disk storage requirements for the object files in the environment is approximately doubled.

4. What time constraints are there?

Object Files

An *object file* partition is formed by linking multiple object files together as one would an executable or shared object, but leaving relocation infomation in the resulting object file. This means the resulting object file can then be linked into another executable or shared object. An object file partition does not need all its external references resolved, nor does it need to define a main routine.

Link Options

Concurrent C/C++ supports a set of link options for each partition. These link options are persistent and can be specified using the following options to **`c.partition`** :

`-oset` *opts*	Sets the link options as indicated by *opts*
`-oappend` *opts*	Appends the *opts* argument to the link option listing
`-oprepend` *opts*	Prepends the *opts* argument to the link option listing
`-oclear`	Clears the link options

opts is a single parameter containing one or more link options; it must be enclosed in double quotes.

A link option set is maintained for each Concurrent C/C++ partition and these options remain effective throughout the life of the partition. Any changes to these options should be done using **`c.partition`**.

For more information about setting link options with **`c.partition`**, see "Link Options" on page 4-55. Also, see "c.link" on page 4-35 for details about this internal utility.

A list of available link options can be found under "Link Options" on page 4-55 .

Compilation and Program Generation

The compiler operates in several distinct phases, designed to satisfy the needs of the entire software development process. These phases include:

- Syntax checking
- Semantic checking
- Code generation and optimization
- Instruction scheduling
- Machine-code assembly

Various options can be specified with the **`c.options`** command in order to control compilation phases. For example, during preliminary software development, it is often useful to limit the compilation phases to syntax and semantic checking. Errors from these phases can be brought up into a text editor automatically for fast, iterative editing and compiling.

Compilation

Concurrent C/C++ uses an C/C++ compiler that supports the C/C++ language specification as defined in the ISO/IEC 14882 *Programming languages -- C++*.

Automatic Compilation Utility

Concurrent C/C++ provides **`c.build`** for automatic compilation and program generation. **`c.build`** calls various internal tools to create an executable image of the program. See "c.build" on page 4-9 for more information.

Compile Options

Unlike most compilation systems, Concurrent C/C++ uses the concept of *persistent options*. These options do not need to be specified on the command line for each compilation. Rather, they are stored as part of the environment or as part of an individual unit's information. These options are "remembered" when the Concurrent C/C++ compilation tools are used.

There are four "levels" of compilation options:

- Permanent environment-wide options

- Temporary environment-wide options
- Permanent unit options
- Temporary unit options

Environment-wide Options

Environment-wide options apply to all units within that environment. All compilations within this environment then observe these environment-wide options unless overridden.

Environment-wide options can be overridden by

- Temporary environment-wide options
- individual unit compile options (permanent or temporary - see below)
- command-line options (which change temporary options on a unit)
- pragmas in the source of the units themselves

Permanent Unit Options

Each unit has its own set of options permanently associated with it that override those specified for the environment. They may be specified and later modified via the **`c.options`** utility.

See the description of “c.options” on page 4-49 for more details.

Temporary Unit Options

Each unit also has a set of options that may be temporarily associated with it that override those that are permanently associated with it.

- If a unit is manually compiled (using **`c.compile`** - see “c.compile” on page 4-14) with any specified options, these are added to its set of temporary options.
- The temporary options may also be set using the **`c.options`** tool.

Temporary options allow users to “try out” options under consideration. By designating these options as “temporary”, the user can first see the effect these options have and then decide if this is what is desired. If so, Concurrent C/C++ provides a way to add these temporary options to the set of permanent options for that unit using **`c.options`**. If these options are not what the user desires, **`c.options`** also provides a way to eliminate all temporary options from a unit (or from all units in the environment).

Another case in which temporary options might also prove useful is one in which a unit needs to be compiled with debug information. If this is not the manner in which the unit is normally compiled, a temporary option can be set for that unit to be compiled with debug information. When the debug information is no longer needed, the temporary option can be removed and the unit can be recompiled in its usual manner.

See the description of “c.options” on page 4-49 for more details.

Effective Options

These levels have a hierarchical relationship to one another. Environment-wide options can be overridden by permanent unit options which can be overridden by temporary unit options. The set of *effective options* for a unit are that unit's sum total of these three option sets, with respect to this hierarchical relationship. Table 3-1shows an example of a unit's effective options based on the relationship between its environment-wide options, permanent unit options, and temporary unit options.

Table 3-1. Effective options based on hierarchical relationship

Permanent environment-wide options		**-g**	**-O2**	**-Qinline_depth=2**
Temporary environment-wide options				**-Qinline_depth=1**
Permanent unit options	**-!S**		**-O3**	
Temporary unit options	**-S**	**-!g**		
EFFECTIVE OPTIONS	**-S**		**-O3**	**-Qinline_depth=1**

As shown in this example, compilation options can be negated by preceding the option with the "**!**" symbol. Therefore, the option "**-!g**" means no debug information should be generated for this unit. Because it is a temporary option for only this particular unit, all other units in the environment will be compiled with debug information (due to the "**-g**" environment-wide option listed in the example).

Option sets controlling a particular attribute of the compilation override each other. Thus **--no_anachronisms** will override **--anachronisms** and **--!anachronisms** will override any anachronism setting (i.e., both **--no_anachronisms** and **--anachronisms**).

See "Invoking the Compiler" on page 1-13 for a list of available compilation options.

Compilation States

Units in the environment can be in any of several different compilation states:

- `uncompiled`

  The state of a newly-introduced unit, or one that has been invalidated. The environment is aware of the unit and some basic dependency information but very little else.

- `preprocessed`

  In this state, proprocessing is done, so the full set of include files is known, but no parsing has been done.

- `parsed`

  In this state, some semantic information about the unit has been generated. There is a complete picture of the meaning of the unit, but none of the actual implementation. Needed and available instantiatable entities have been determined.

- `prereorder`

  In this state, source file has been compiled, and pseudo-assembly language has been output, but the instruction scheduler has not been run.

- `assembly`

  In this state, The pseudo-assembly language output has been run through the instruction scheduler, and an assembly file has been generated.

- `compiled`

  Object files have been generated for the unit

The benefit of having this information generated at each of these states for each unit in the environment is that it allows the compilation utility to use this information to produce better code in the unit currently being compiled.

`c.build` allows the user to compile units to a specified state using the **`-state`** option, however, `compiled` is the only fully supported state allowed for this option in the current release. See "c.build" on page 4-9 for more information.

NOTE

Only the `uncompiled` and `compiled` states are available at this time. These states are documented because they are visible in such utilities as **`c.build`**, **`c.compile`**, and **`c.ls`**.

Consistency

Along with compilation states comes the idea of *consistency*. Each unit is considered consistent up to a particular state. This means that it is valid *up to that state of compilation*. Any recompilation of the unit can start from that state. It does not need to go through the earlier stages of recompilation.

Modification of a unit may possibly change its consistency. Modifications include:

- changes to the source file itself
- changes to any of the options
- changes to any required units upon which this unit depends

For example, if the source of a unit has been modified since it was last compiled, the semantics of the unit are potentially changed. New semantic information about the unit must be generated. Therefore, it is considered "consistent up to the `uncompiled` state". This means that when it is recompiled, it must start at the inconsistent state, `uncompiled`.

Not all changes to a unit make it “consistent up to the `uncompiled` state”. Changing the options on a unit may not affect the syntax or semantics of a unit and therefore do not require a total recompilation.

Programming Hints and Caveats

In general, programs that are to be debugged with NightView should not be optimized. Optimization levels `GLOBAL`, `MAXIMAL`, and `ULTIMATE` should be reserved for thoroughly tested code.

There is no misaligned handler. The hardware allows misaligned integer (fixed-point) data accesses, but floats and long floats must be word-aligned. There is a performance penalty for misaligned accesses.

Linking Executable Programs

Concurrent C/C++ provides a linker that verifies and creates an ELF executable image of all component units required for a given main unit. The linker can be invoked directly but should be called from the compilation utility **`c.build`**.

Debugging

Real-Time Debugging

In addition to the symbolic debugging capabilities provided by NightView (see *NightView User’s Guide* (0890395)), and the post-analysis debugging capabilities provided by the tracing mechanism, Concurrent C/C++ also provides several ways to debug programs in real-time.

Debug Information and cprs

The **`cprs`** utility (see **`cprs(1)`**), supplied with PowerMAX OS, reduces the size of an application by removing duplicate type information. The Concurrent C/C++ compiler reduces the value of this tool by already referencing the debug information for types defined in other units from those other units. However, the **`cprs`** utility can still reduce the size of Concurrent C/C++ applications. Also, if debug code from other languages is included in an application, then **cprs** can significantly reduce the size of those portions as well.

If users compile only certain units with full debug information, it is possible to produce duplicate debug information for types in several units. Also, even if an entire application is compiled with full debug information, anonymous types are frequently duplicated in several units, as are types for certain compiler-generated constructs.

Source Control Integration

There are a number of software packages for managing versions of source code. The environment provides a rudimentary way inter integrate with such packages. In the directory **/usr/ccs/release/***release***/source**, the system administrator can create a directory for a particular source management system, that we'll call *sms* for illustrative purposes. In that directory, the system administrator should place two scripts: pre-edit and post-edit. These scripts are destined to run before and after a source file is edited.

Now, the user, when he creates his environment, would issue the **c.mkenv** command with the -src option like this:

```
$ c.mkenv -src "sms -v %f"
```

Now, whenever **c.edit**, or any command that invokes **c.edit** is run, the pre- and post-edit scripts are run before and after the editor, passing the -v option and the name of the source file, as if the user had invoked:

```
$ /usr/ccs/release/release/source/sms/pre-edit \
        -v file.c
$ $EDITOR file.c
$ /usr/ccs/release/release/source/sms/post-edit \
        -v file.c
```

DISCLAIMER

Concurrent Computer Corporation does not support any particular source management system. The **-src** mechanism may not provide the flexibility needed by any particular system.

Makefile Integration

The C/C++ Program Development Environment provides a mechanism to escape to Makefiles or any other arbitrary software to generate source files by means other than editing them. The **c.intro** command includes a **-make** option that specifies an arbitrary shell command line that will be used to construct the source file whenever it needs to determine if it needs to rebuild the unit that is being introduced.

```
$ c.intro -make "make myfile.c" myfile.c
```

The contents of the makefile then might be something like this:

```
myfile.c: definition.txt
          c.build builder
          builder <definition.txt >myfile.c
```

Whenever **`c.build`** needs to build the unit myfile, it will invoke the make command. If the definition file for **`myfile.c`** is newer than **`myfile.c`**, it will in turn, recursively invoke **`c.build`** to make sure the builder tool is there, then invoke it to build the source file.

If invoking the make command changes the timestamp on **`myfile.c`**, then the unit myfile will be rebuilt. Otherwise, the existing object for unit myfile will be used.

The make command may be changed by using the **`c.options`** command:

```
$ c.options -make “make myfile.h myfile.c” -source myfile.c
```

When building a partition, all the make commands needed by any dependant unit of the partitition are run before any units are built. Thus it is only necessary for one source file to do the make that would, for example, construct a header file needed by many compilations.

4
Program Development Environment Utilities

4 Program Development Environment Utilities

The Concurrent C/C++ Program Development Environment consists of a number of utilities that provide support for library management, compilation, program generation, and debugging. Table 4-1 lists these tools and gives a brief description of each one.

Table 4-1. Concurrent C/C++ Utilities

Environment Utilities	
c.mkenv	Create an environment which is required for compilation, linking, etc.
c.path	Display or change the Environment Search Path for an environment
c.options	Set compilation options for the environment (or for units)
c.rmenv	Destroy an environment; compilation, linking, etc. no longer possible
c.chmod	Modify the UNIX file system permissions of an environment
c.release	Display release installation information
c.script	Generate a script that will recreate an environment
c.freeze	Disallow changes to, and optimize uses of an environment
c.make	Generate a Makefile to reproduce build the partitions in the environment
c.restore	Restore a corrupted database from an automatic backup
Unit Utilities	
c.ls	List units in the environment (state, source file, dependencies, etc.)
c.options	Set compilation options for units (or the environment)
c.edit	Edit the source of a unit, then update the environment
c.cat	Output the source of a unit
c.touch	Make the environment consider a unit consistent with its source file's timestamp
c.invalid	Force a unit to be inconsistent thus requiring it to be recompiled
c.instantiation	Control instantiation automation
c.fetch	Fetch the compiled form of a unit from another environment
c.expel	Expel fetched or naturalized units from the environment

Table 4-1. Concurrent C/C++ Utilities (Cont.)

Source File Utilities	
`c.intro`	Introduce source files (and units therein) to the environment
`c.lssrc`	List sources files in the environment
`c.rmsrc`	Remove knowledge of source files (and units therein) from the environment
`c.grep`	Search the source files known to the environment
Debug Utilities	
`c.analyze`	Optimize or analyze performance of fully-linked executables
`c.report`	Generate profile reports in conjunction with **`c.analyze -P`**
`c.demangle`	Transform mangled names back to C++ names with signatures
Compilation Utilities	
`c.build`	Compile and link as necessary to build a unit, partition or environment
`c.partition`	Define or display a partition for the linker
Internal Utilities	
`c.install`	Install, remove, or modify a release installation
`c.compile`	Compile the specification and/or body of one or more units
`c.prelink`	Resolve unit selection and template instantiation
`c.error`	Process diagnostic messages generated by the compiler and other tools
`c.link`	Link a partition (an executable, archive or shared object file)
`c.features`	List features added in this release (for NightBench's use)
Help Utilities	
`c.help`	List usage and summary of each Concurrent C/C++ utility
`c.man`	Invoke/position interactive help system (requires an X terminal)

The following sections provides an overview of each of the utilities. For easy reference, the command syntax and options available for each utility are provided in tabular format. Available options for each tool are also provided by specifying the **`-H`** option to any tool listed.

Common Options

There are a number of options that are the same for each utility. They are listed for each tool but are also listed below.

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-H`	help	Display syntax and options for that particular function
`-V`	verify	Show actions that would be executed without actually doing them. Not all commands have a **`-V`** option.
`-v`	verbose	Provide feedback as actions are performed. Not all commands have a **`-v`** option.
`-vv`	very verbose	Provide extra feedback as actions are performed. This usually includes showing the exact command line used to invoke subordinate tools. Not all commands have a **`-vv`** option.

unit-id is defined by the following syntax:

unit **`| all`**

See "Unit Identification" on page 3-4 for more information about the *unit-id*.

c.analyze

Optimize or analyze performance of fully-linked executables

The syntax of the **`c.analyze`** command is:

`c.analyze [`*options***`]`** *executable-file*

The following represents the **`c.analyze`** options:

Option	Meaning	Function
`-A`	all	Include all the routines in the analysis (initial default)
`-a` *routine*	add	Add the specific named *routine* to the list of routines to analyze (implies **`-N`**)
`-C`	cache	Gather cache activity statistics during profiling. Requires **`-P`** option.
`-D` *flag*	debug	Turn on the specified debug flag. Not of general user interest. Use **`-Dhelp`** for list of options
`-Dhelp`	debug help	List of debug options
`-d` *file*	disassemble	Generate a detailed disassembly listing of each routine included in the analysis in *file*. The listing is done on a per basic block basis. By default this only generates the assembler listing, the clock cycle each instruction executes at (relative to the beginning of each basic block), and the reason any instruction is delayed. Use the **`-v`** option for more detail. Use **`-Zstage_status`** for much more verbose status of each pipeline stage each cycle. Use **`-`** for *file* to direct output to **`stdout.`**
`-env` *env*	environment	Specify an environment pathname. Defaults to current directory.
`-g` *file*	global	Generate global program statistics to *file.* Use **`-`** for *file* to direct output to **`stdout`**
`-H`	help	Display syntax and options for this function
`-i`	information	Display information only messages
`-N`	null	Set the list of routines to be analyzed to the empty set (no routines)
`-n`	nesting level	use nesting level to weight the count of lis instructions. This option is used with the -O option.
`-O` *file*	optimize	Generate a new program file in *file* which has been optimized by replacing many of the two-instruction sequences (which are required to reference global memory locations) with single instructions which use the reserved linker registers as base registers. This allows faster access to the four most commonly referenced 64K data blocks. Certain library routines that are known to access the linker registers (e.g., `setjmp` and `longjmp`) are automatically excluded from the optimization process. The **`-X`** option may be used to specifically exclude others. (Normally any reference to a linker register will cause an error)

Option	Meaning	Function
-P *file*	profile	Generate a new program file in *file* which has been patched to gather profiling statistics on each basic block and dump them to *file*.`prof` on exit. The **c.report** program can be used to generate various reports from this information. The **-X** option may be useful with this option.
-r *file*	routine	Print summary statistics for each routine to *file*. Use - for *file* to direct output to **stdout**
-rel *release*	release	Specify a Concurrent C/C++ release (other than the default release)
-S *section*	section	Analyze the specified *section* of the object file, rather than the default **.text** section
-s *routine*	subtract	Subtract the *routine* from the list to be analyzed. Implies the **-A** option.
-vv	very verbose	Print invocations of sub-processes
-v	verbose	Show verbose timing info in disassembly listing
-W*routine***[**=*weight***]**	weight	Specify a weighting factor for counting lis instructions in specified routine. If weight is omitted, 5 is assumed. This option is used with the -O option.
-w	warnings	Suppress the output of warning messages
-X *routine*	exclude	Declare *routine* to be the name of a subroutine which causes the program to exit. When the **-P** option is used, this routine, when called, will append the accumulated statistics to the **.prof** file. After writing the statistics data set to the **.prof** file, the statistics are reset to zero. When the **-O** option is used, the **-X** option will exclude the named routine from the optimization
-Zraw_names	raw	Print routine and source file names in raw form (i.e. do not filter)
-Z*misc*	keyword	Set various obscure keyword flags (use **-Zhelp** for list)
-Zhelp	keyword help	Displays list of obscure keyword flags

NOTE

The **c.analyze** command is not normally invoked by the user, except to do profiling; it is most often called by **c.link** (which is called in turn by **c.build**).

NOTE

The **-a**, **-s**, and **-X** options to **c.analyze** take a routine name as a parameter. The **c.analyze** processor recognizes C/C^{++} routines only by their link names. These names may not be intuitive for C++ routines. Using the **nm(1)** utility may be helpful in order to determine C++ routine names.

The **c.analyze** tool is available for performing static performance analysis of C/C^{++} object files. **c.analyze** reads the object, finds the routine entry points, breaks the routines into basic blocks, and analyzes each basic block for instruction times. **c.analyze** can generate detailed basic block information or a flow graph picture showing the whole program. By default, all routines are analyzed, but the above options can be used to control which routines are included or excluded.

With the **-O** option, **c.analyze** generates a new program file that optimizes many double word memory reference instructions into single words by use of the linker registers.

With the **-P** option, **c.analyze** generates a new program file that will accumulate profiling statistics. Running this program file generates profiling data that can be used with the **c.report** command to provide profiling statistics.

Link-Time Optimizations with c.analyze

To enhance the optimization of C/C^{++} source, in addition to compiling the source code at the MAXIMAL level (**-O3**), you can elect to invoke **c.analyze** when linking your C/C^{++} programs in order to perform additional optimizations at link time. For example, the **-O** option to **c.analyze** replaces many of the two-instruction sequences required for referencing global memory locations with a single instruction.

You can invoke **c.analyze** in two ways: either directly on executables or as an option to the linker (**c.link**).

To invoke the **c.analyze** optimizer directly on an executable file (**a.out**), simply type the following:

```
$ c.analyze -O na.out a.out
```

The original executable, **a.out**, remains the same and the resulting executable generated by **c.analyze** is contained in a file called **na.out**.

Alternatively, you can invoke **c.analyze** at link time by specifying the **-O** link option for a given partition:

```
$ c.partition -oappend -O main
$ c.build main
```

What results from this sequence of commands is that a single executable file (**a.out**) is optimized at level GLOBAL followed by an additional link-time optimization performed by the **c.analyze** optimizer.

Because of the **-O** option, **c.analyze** performs the following link optimization. It replaces the two-instruction sequences (which are required to reference global memory locations) with single instructions which use the reserved linker registers (r28 and r30) as base registers. This allows faster access to the two most commonly referenced 64K data blocks.

NOTE

Certain library routines that are known to access the linker registers (e.g., setjmp and longjmp) are automatically excluded from the optimization process.

Additional **c.analyze** options may be specified directly on the **c.analyze** command line or indirectly by supplying an option string via the **-WA** link option for a given partition.

Profiling with c.analyze

In addition to performing link-time optimizations, **c.analyze** can be used in tandem with the **c.report** tool in order to generate profiling statistics. See “c.report” on page 4-61.

To profile an executable C/C^{++} program with **c.analyze**, the **-P** (profiling) option must be specified. With the **-P** option set, a new executable file is created that has been patched to gather profiling statistics. The original executable file remains intact. For example, the following command line:

```
$ c.analyze -P  profiled_a.out  a.out
```

takes the executable **a.out** as input, profiles it, and then produces the patched executable file **profiled_a.out**. The original executable remains unchanged; however, invoking the patched executable gathers profiling information and dumps this information to the file **profiled_a.out.prof**. The **.prof** file can then be displayed in various formats with the help of the **c.report** program.

Many other options are available for profiling executables using **c.analyze**. Refer to the online man pages for more information about **c.analyze** and **c.report**.

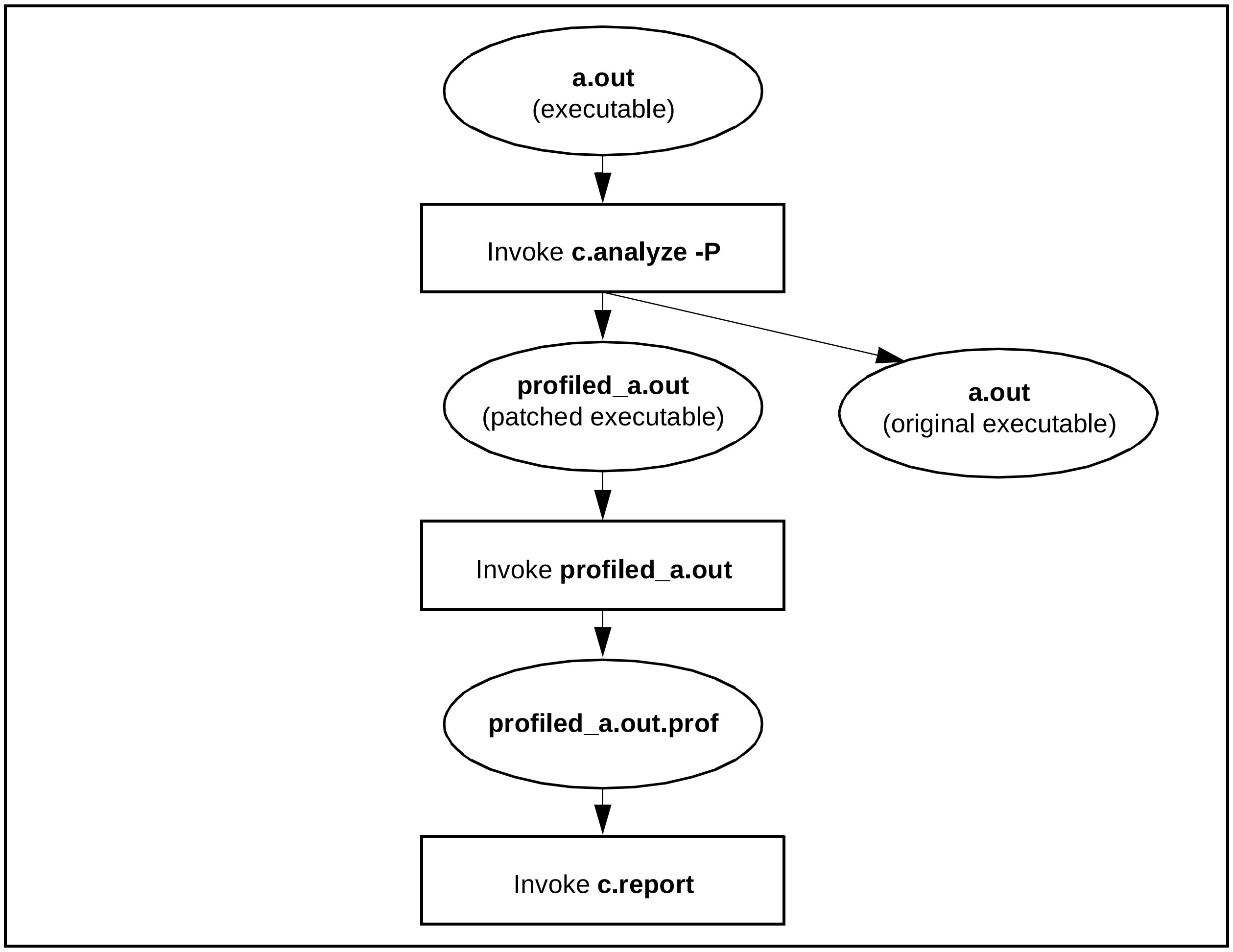

Figure 4-1. Profiling a Program

c.build

Compile and link as necessary to build a unit, partition or environment

The syntax of the **`c.build`** command is:

`c.build [`*options***`] [`***partition*** `...]`**

The following represents the **`c.build`** options:

Option	Meaning	Function
`-allparts`	all partitions	Build all partitions in the environment. This option is not allowed if the **`-o`** option is specified.
`-C` *"compiler"*	compiler	Use *compiler* to compile units (may be used to pass options to the compiler, e.g. **`c.build -C "c.compile -v"`**)
`-e[e|l|v]`	error	Pipe compiler output through **`c.error`**: **`-e`** lists errors to stdout; **`-ee`** embeds errors in the source file and invokes `$EDITOR`; **`-el`** lists errors with the source file to **`stdout`**; and **`-ev`** embeds errors in the source file and invokes **`vi`**.
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-i`	infos	Suppress **`c.build`** information messages
`-L`*"linker"*	linker	Use "*linker*" to link partitions (may be used to pass options to the **`c.link`**)
`-link`	link only	Link an non-archive without updating any dependant partition. For internal use only.
`-noimport`	no import	Don't naturalize foreign units that are not up to date. Generates an error instead.
`-nomake`	no make	Don't invoke make command to build source file. Use whatever version of the source that is already built.
`-nosource`	no source	Skip checks of the source timestamps for out-of-date units (should only be used if no source files have changed)
`-o` ***`file`***	output	Override the output file for the partition being built. Only a single partition file name is allowed with this option.
`-P`*"prelinker"*	prelinker	Use *"prelinker"* to prelink partitions (may be used to pass options to **`c.prelink`**).
`-part` *partition*	partition	Build the given *partition*, all included units and all units upon which they directly or indirectly depend
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)

Option	Meaning	Function
-source *file*	source file	Build all units defined in the given source *file* and all units upon which they directly or indirectly depend
-state *s*	state	Build all specified units to compilation state *s*.
-stop	stop on errors	If an error is encountered, stop building (normally, any units not dependent upon the erroneous units would be built)
-u *"unit list"*	unit	Compile the unit. The *unit* parameter can be "`all`".
-ufile *file*	units from file	Build the units listed in *file*
-V	verify	List compilations that would occur, but do not actually perform them
-v	verbose	Display actions as they are done
-vv	very verbose	Display commands as they are done
-w	warnings	Suppress **c.build** warnings

NOTE

Specified partitions are equivalent to partitions passed as arguments to the **-part** option.

If no options are specified, then all units and partitions in the environment are built.

Concurrent C/C++ provides the **c.build** utility to build partitions and units in an environment. **c.build** determines which units must be compiled to build the given target, builds them, and calls the linker to produce the desired partition. **c.build** examines the current environment (and the environments on the Environment Search Path), determines and automatically executes the proper sequence of compilations and links necessary to build the given partition.

Targets to **c.build** can be:

partitions	which can be specified directly, with the **-part** option, or with the **-allparts** option
units	which can be specified the **-u** option

If the **-u** option is specified, **c.build** ensures the named *unit* is up-to-date.

Normally, **c.build** attempts to build all units in the current Concurrent C/C++ environment and all units on the Environment Search Path that are required. The **-noimport** option can be used to prevent automatic recompilation of out-of-date units from other environments.

When **c.build** invokes the "make" commands attached to source files (see "c.intro" on page 4-32) it sets the environment variable `PDE_BUILD_OPTIONS` to the option set it was invoked with. This is so that the "make" commands may pass these options on to **c.build** invokations that it might make. For example,

```
c.intro -make ‘make BUILD=”$PDE_BUILD_OPTIONS” table.c’ table.c
```

And the Makefile contains:

```
table.c: configuration.txt
     c.build $(BUILD) build_table
     ./build_table < configuration.txt > table.c
```

If these options aren’t passed on, then NightBench will not see the activity of such recursive **`c.build`** invocations. Options thatwould not be appropriate for recursive **`c.build`** invokations (such as **`--allparts`**) are not included in `$PDE_BUILD_OPTIONS`.

See “Compile Options” on page 3-9 and “Link Options” on page 3-8 for more information.

c.cat

Output the source of a unit

The syntax of the **`c.cat`** command is:

`c.cat [`*options***`]`** *unit-id*

The following represents the **`c.cat`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-h`	no header	Does not output filename header
`-l`	line numbers	Prepend each line of source with its line number
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)

unit-id is defined by the following syntax:

unit

ie, **`all`** may not be specified as a unit name.

The **`c.cat`** command is similar to the UNIX **`cat(1)`** command in functionality. It accepts as its argument a *unit_id* and prints to **`stdout`** the source file in which this unit is found.

By default, it outputs a header containing the full path name of the source file. This can be suppressed by specifying the **`-h`** option.

Also, line numbers can be prepended to each line of source by using the **`-l`** option.

c.chmod

Modify the UNIX file system permissions of an environment

The syntax of the **`c.chmod`** command is:

`c.chmod [`*options***`]`** *access_mode*

The following represents the **`c.chmod`** options:

Option	Meaning	Function
`-a`	all	In addition to internal environment files, change the permissions of all files associated with the environment, including source files and partition targets
`-env` *env*	environment	Specify an environment pathname
`-f`	force	Force, if some environment components are missing
`-H`	help	Display syntax and options for this function
`-i`	ignore	Quietly ignore all non-fatal errors
`-include`	`#include` files	Change permissions on primary source files and any files they `#include` (except system include files). Units built from the primary source file must have been compiled in order for the environment to have discovered what files were `#include`d.
`-p`	partitions	Change permissions on partitions in the environment
`-q`	query	Display the permissions on the current environment
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-s`	source	Change permissions on primary source files (see also **`-include`**).

access_mode is the permissions as would be specified using the **`chmod(1)`** command (q.v.).

c.compile

Compile one or more units

INTERNAL UTILITY

This tool is used internally by **`c.build`** which is the recommended utility for compilation and program generation.

`c.compile` is not intended for general usage.

The syntax of the **`c.compile`** command is:

`c.compile [`*options***`] [--] [`***compile_options***`] [`***unit-id*** **`...]`**

The following represents the **`c.compile`** options:

Option	Meaning	Function
`-e[e|l|v]`	error	Pipe compiler output through **`c.error`**: **`-e`** lists errors to stdout; **`-ee`** embeds errors in the source file and invokes `$EDITOR`; **`-el`** lists errors with the source file to **`stdout`**; and **`-ev`** embeds errors in the source file and invokes **`vi`**.
`-env` *env*	environment	Specify an environment pathname, default is $PWD
`-H`	help	Display syntax and options for this function
`-HC`	help compile	Display list of compile options
`-HQ`	help qualifier	Display list of qualifier keywords (**`-Q`** options)
`-language` *lang*	language	Select **`C`** or **`C++`** as language (*lang*) to be used.
`-partition` *part*	partition	Let compiler know which partition is being built for template instantiation purposes (not necessary, but it lets the compiler make better automatic decisions)
`-quiet`	quiet options	Suppress display of effective options
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-state` *s*	state	Compile the specified unit to compilation state *s*.
`-v`	verbose	Print header for each compilation
`-vv`	very verbose	Print subordinate tool invocations

unit-id is defined by the following syntax:

unit **`| all`**

If *compile_options* are specified to this command, they override the set of temporary unit options. For instance, if the temporary compile options for the unit `hello` consist of **`-S`** and the following command is issued

```
$ c.compile -g hello
```

the effective options will now consist of **`-S`** and **`-g`**.

See “Link Options” on page 4-70 for list of compile options.

c.demangle

Output the source form of a unit name

The syntax of the **`c.demangle`** command is:

`c.demangle [`*options***`]`**

The following represents the **`c.demangle`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)

The **`c.demangle`** utility is a filter that accepts mangled Concurrent C++ symbol names, such as those found in a C++ object file, and returns C++ unit names in expanded form. It reads text from **`stdin`**, transforms anything that looks like a mangled name and writes the result to **`stdout`**. It is useful for processing the output from utilities such as **`nm(1)`**, **`dump(1)`**, etc.

c.edit

Edit the source of a unit

The syntax of the **`c.edit`** command is:

`c.edit [`*options***`]`** *unit-id*

The following represents the **`c.edit`** options:

Option	Meaning	Function
`-e` *editor*	editor	Use *editor* instead of `$EDITOR`
`-env` *env*	environ-ment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-i`	inhibit	Do not immediately notify the environment that the unit has changed
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-src “`*dir* **`[`***options***`]` `%f”`**	source management	Run pre- and post-edit scripts for interfacing with source management.
`-v`	verbose	Generate verbose output.

unit-id is defined by the following syntax:

unit **`| all`**

c.error

Process diagnostic messages generated by the compiler and other tools

INTERNAL UTILITY

This tool is used internally by **`c.build`** which is the recommended utility for compilation and program generation.

`c.error` is not intended for general usage.

The syntax of the **`c.error`** command is:

`c.error [`*options***`]`**

The following represents the **`c.error`** options:

Option	Meaning	Function
`-e [`*editor***`]`**	editor	Embed error messages in the source file and invoke the specified *editor*. The default editor is `$EDITOR`.
`-env` *env*	environment	Specify an environment pathname
`-f` *file*	source file	Restrict errors to those in specified source file
`-H`	help	Display syntax and options for this function
`-l`	listing	Produce listing to **`stdout`**
`-N`	no line #'s	Do not display line numbers
`-o`	order	Do not sort the order of the diagnostics by file and line number; process each diagnostic in the order given
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-s`	suppress	Suppress non-error lines
`-src "`*dir* **`[`***options***`]`** **`%f"`**	source management	Run pre- and post-edit scripts for interfacing with source management.
`-vi`	**`vi`**	Embed error messages in the source file and invoke the **`vi`** editor
`-W`	warnings	Ignore warnings

Compiler output may be redirected into a file and examined with the aid of the **`c.error`** command or can be piped directly into **`c.error`** via the **`-e`** option to **`c.build`**.

`c.error` reads the specified file or the standard input, determining the source file(s) containing errors and processing the errors according to the options given.

Perhaps more generally useful are the **`-e`** options to **`c.build`** (**`-e`**, **`-ee`**, **`-el`**, **`-ev`**) , which automatically call **`c.error`** to process any compiler error messages resulting from the current compilation.

The following code comprises the file **`badtry.c`**. This file (which intentionally contains errors) is used to illustrate various ways Concurrent C/C++ tools can use **`c.error`** to process error messages.

```
#include <stdio.h>
main() {
   for (inst i=99; i>0; i--) {
      printf("%d bottles of soda on the wall\n"
             "     %d bottles of soda on the wall\n"
             "if one of those bottles should happen to fall\n"
             "     %d bottles of soda on the wall\n",
             i,i,i-1);
   }
}
```

Before it can be compiled, the file must be introduced into a Concurrent C/C++ environment, and a partition must be created for it:

```
$ c.mkenv
$ c.intro badtry.c
$ c.partition -create executable badtry
```

The file can be compiled and the output directed as follows (**`stdout`** is redirected to the file **`badtry.errors`**):

```
$ c.build 2> badtry.errors
```

Screen 4-1 shows the contents of file **`badtry.errors`**.

```
"badtry.c", line 3: error: identifier "inst" is undefined
     for (inst i=99; i>0; i--) {
          ^

1 error detected in the compilation of "badtry.c".
ec++: ERROR: Errors in the cxc++ pass, status = 2
c.compile: failed to compile unit badtry
c.build: failed building partition badtry
c.build: there was a failure building one or more partitions
```

Screen 4-1. File badtry.errors

This file can simply be listed, if desired, but it is more useful to use **`c.error`** as follows.

```
$ c.error  -l  badtry.errors
```

outputs the listing that appears in Screen 4-2.

```
Non-specific diagnostics:
 ec++: ERROR: Errors in the cxc++ pass, status = 2
 c.compile: failed to compile unit badtry
 c.build: failed building partition badtry
 c.build: there was a failure building one or more partitions

*********************** badtry.c ******************************


  1:#include <stdio.h>
  2:main() {
  3:   for (inst i=99; i>0; i--) {
A -----------^
A:error:identifier "inst" is undefined
  4:     printf("%d bottles of soda on the wall\n"
  5:       "     %d bottles of soda on the wall\n"
  6:       "if one of those bottles should happen to fall\n"
  7:        "     %d bottles of soda on the wall\n",
  8:        i,i,i-1);
  9:   }
  10:}
```

Screen 4-2. c.error -l Output Listing

With the **`-v`** option, **`c.error`** writes the error messages directly into the original source file and calls the **`vi`** text editor. Line numbers are suppressed, and error messages marked with the pattern ###.

After the compilation,

```
    $ c.error  -v  <  badtry.errors
```

calls **`vi`**.

The ### is provided so that error messages can be easily found and subsequently deleted. For example, if invoked with the **`-v`** (**`vi`**) option, **`c.error`** embeds error text in the source file and then invokes the **`vi`** editor. All error text can easily be found and removed with simple editor commands by searching for the ### pattern and deleting. In **`vi`**, for instance, the sequence “**`:g/###/d`**” deletes all lines matching the ### pattern.

It should also be noted that all error message lines are prefixed with `//`, which denotes an C++ comment. Thus, even if **`c.error -v`** has been used to intersperse error messages into a file, the compiler can still process that file without deleting the error messages. Since **`-v`** places the error messages directly in the source file, if **`c.error -v`** is called again before the messages are deleted and the error corrected, a second copy of the same messages appears.

The file **`badtry.c`** can now be edited to repair the error and resubmitted to the compiler. If those errors are fixed correctly, semantic analysis can proceed.

The preferred method for achieving the same results is to invoke **`c.build`** with a **`-e`** option (**`-ev`** is used in the following example).

```
    $ c.build -ev
```

Now, when errors are encountered during compilation, the **`vi`** editor will be automatically opened to the source file with the error messages embedded in it. Also, upon leaving the editor, the compiler offers to recompile the file.

This method is generally faster for rapid interactive program development because it does not require any intermediate files.

c.expel

Expel fetched or naturalized units from the environment

The syntax of the **`c.expel`** command is:

`c.expel [`*options***`]`** *unit-id* **`...`**

The following represents the **`c.expel`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-v`	verbose	Print message for each expelled unit

unit-id is defined by the following syntax:

unit **`| all`**

Local versions of foreign units may be created via the **`c.fetch`** tool (see “c.rmsrc” on page 4-65). These versions are called *fetched*. (See “Nationalities” on page 3-4 for a more detailed discussion.)

It may be desirable to later remove these local versions, thus making the foreign versions once again visible. The **`c.expel`** tool is provided for this purpose.

NOTE

Other methods exist for removing native units. See Section “c.rmsrc” on page 4-65 for more information.

c.features

Indicate features of current release

INTERNAL UTILITY

This tool is used internally by **`NightBench`**.

`c.features` is not intended for general usage.

The syntax of the **`c.features`** command is:

`c.features [`*options***`]`**

The following represents the **`c.features`** options:

Option	**Meaning**	**Function**
`-env` *env*	environment	Specify an environment pathname, default is $PWD
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)

The **`c.features`** command outputs a list of features added or altered since the initial release. This is so NightBench can tailor its graphical interface to the release of tools being used without having to be congizant of specific release names.

c.fetch

Fetch the compiled form of a unit from another environment

The syntax of the **`c.fetch`** command is:

`c.fetch [`*options***`]`** *unit-id* **`...`**

The following represents the **`c.fetch`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-from` *env*	from env	Specify an environment pathname from which to fetch the unit(s)
`-H`	help	Display syntax and options for this function
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-v`	verbose	Display a message for each fetched unit

unit-id is defined by the following syntax:

unit **`| all`**

NOTE

If the **`-from`** option is not specified, **`c.fetch`** will try to "find" the specified unit by searching the Environment Search Path.

At times, it may be desirable for users to be able to force copies of specified units from other environments into the current environment. This command will cause the specified foreign units to be built in the local environment as if they were introduced as local units.

The **`c.expel`** tool is provided to allow a fetched unit to be removed from the local environment, thus restoring visibility to the foreign version. See "c.expel" on page 4-22.

c.freeze

Freeze an environment, preventing changes

The syntax of the **`c.freeze`** command is:

`c.freeze [`*options***`]`**

The following represents the **`c.freeze`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-q`	query	Displays an environment's frozen status
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-t`	transitive	Freeze whole Environment Search Path
`-u`	unfreeze	Thaw the environment, allowing changes
`-v`	verbose	Displays the environment(s) being frozen (or thawed)

An environment may be frozen using the **`c.freeze`** utility. This changes an environment so that it is unalterable.

Any environment which will not be changed for a significant period of time and which will be used by other environments is a good candidate to be frozen to improve compilation performance.

c.grep

Grep the introduced source in an environment

The syntax of the **c.grep** command is:

> **c.grep [***options***]** *expression* **[***pattern***]**

The following represents the **c.grep** options:

Option	Meaning	Function
-b	block number	Precede each line with block number on which *expression* was found
-c	count	Print only count of lines on which *expression* was found
-E	Expression	Treat all specified expressions in *expression* and *exprfile* as full regular expressions. See **egrep(1)** for an explanation of full regular expressions
-e *expression*	expression	Specify regular expression or strings in the command line
-env *env*	environment	Specify an environment pathname
-F		Treat all specified expressions in *expression* and *exprfile* as character strings. This is the same as invoking **fgrep(1)**
-f *exprfile*	file	Specify file containing *expression*s, one per line
-H	help	Display syntax and options for this function
-i	ignore	Ignore case distinction when matching
-l	list files	List names of flies with matching lines, one per line
-n	number lines	Precede each line by line number in the file
-no_include	no include	Exclude `#include` files
-no_usr_include	no /usr include	Exclude `#include` files with paths beginning with **/usr**
-plist *list*	partitions	Specify a list of partitions whose source is searched
-q	quiet	Do not write any matches to stdout, but exit with zero status if any input line matched.
-rel *release*	release	Specify a Concurrent C/C++ release (other than the default release)
-s	suppress	Suppress error messages about nonexistent or unreadable files
-slist *list*	sources	Specify a list of source files to search
-t	transitive	Include dependendant partitions when searching the source of a partition. Requires that the **-plist** option be used.
-ulist *list*	units	Specify a list of units whose source is searched
-v		Print all lines except those which match *expression*
-x		Print only matches for which *expression* matches the entire line.

The **`c.grep`** command allows the user to invoke **`grep(1)`** on the source files the environment is aware of. Options marked with an asterisk in the table above are passed on to the **`grep(1)`** tool. See that tool's man page for more details regarding them.

There are many ways to specify sources files to be included or excluded from the search. By default, all source files known to the environment, including `#include` files, are included in the search. If any option to specify particular source files is specified, then only those files are searched. Multiple options may be combined to specify source files.

To search the source of particular units, use the **`-ulist`** option. If a unit has not been compled yet, then the environment only knows about its primary source file, so its include files won't be searched. Once the unit has been compiled, the environment knows all the `#include` files that were used it compiling it; so those are searched too.

The user may also specify primary source files instead of units using the **`-slist`** option. If the unit(s) built from that primary source file have been compiled, then all the `#include` files used in compiling all the units are included in the search. Remember that different units compiled from the same primary source file may `#include` different files because of compile options. The **`c.grep`** tool will search all the `#include` files of all the units built from the specified primary source file(s).

The user may also specify partition names using the **`-plist`** option. This will the source of those units upon which a partition directly links with. If the partition includes a unit with the transitive closure specification (e.g., **`-add`** *my_unit*`!`) and the partition has not been built yet, then the environment hasn't determined what units are included in the transitive closure. Only the named unit's source will be searched. However, if the user has compiled such a partition, then all the units included in the transitive closure will be included. If the **`-t`** option is also specified, then dependant partitions, and transitively, their dependant partitions, are also included.

To restrict, rather than add, to the list of files searched, a shell wildcard pattern may be specified on the command line that much match the filenames to be searched.

The user may also restrict the search from searching `#include` files. Either all `#include` files, using the **`-no_include`** option, or system `#include` files, using the **`-no_usr_include option`**, may be excluded from the search. System `#include` files are any `#include` files whose full paths start with **`/usr/`**.

c.help

List usage and summary of each Concurrent C/C++ utility

The syntax of the **`c.help`** command is:

```
c.help
```

c.install

Install, remove, or modify a release installation

The syntax of the **`c.install`** command is:

`c.install -rel` *release* **[***options***]**

The following options are available with the **`c.install`** command:

Option	Meaning	Function
`-arch` *arch*	default arch	Set the default architecture (nh, moto, synergy, etc.) being targeted. Currently only valid under the PLDE, although a future PowerMAX OS release may support cross compilation.
`-d`	default	Mark the selected release installation as the system-wide default
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-i` *path*	install	Install the release located at *path* into the release database (the name is determined from the **`-rel`** option)
`-m` *path*	move	Move the selected release installation to *path*
`-osversion` *osversion*	default os version	Set the default version of PowerMAX OS being targeted. Currently only valid under the PLDE, although a future PowerMAX OS release may support cross compilation.
`-p`	pre-5.1	Mark the selected release isntallation as the default for **`cc`**, **`hc`**, **`cc++`**, and **`c++`**.
`-r`	remove	Remove the specified release installation from the release database
`-rel` *release*	release	Specify a Concurrent C/C++ release (REQUIRED)
`-target` *cpu*	default cpu	Set the default *cpu* being targeted. Defaults to the native cpu under PowerMAX OS. A default must be set under PLDE.
`-v`	verbose	Report changes as they are made

NOTE

Only the System Administrator (or a super user) can invoke **`c.install.`**

The **`-i`**, **`-m`**, and **`-r`** options may never be used together.

The **`c.install`** utility is the tool that allows users to register installations with the system's Concurrent C/C++ database. It may be used to install, move, and remove installations.

When the **`-i`** option is given, then the Concurrent C/C++ structure located at the specified path name is registered with the database as a valid installation. The name of the installation is registered as the release given by the **`-rel`** option. Therefore, the **`-rel`** option is required when using the **`-i`** option to install a Concurrent C/C++ installation.

For example, the following command:

```
$ c.install -rel newc -d -i /somedir/c_dir
```

assumes that **`/somedir/c_dir`** contains a valid Concurrent C/C++ directory structure and "installs" this version of Concurrent C/C++ in the database as **`newc`**.

When the **`-d`** option is used, then **`c.install`** registers the installation with the database, and also marks the installation as the system-wide default installation (as in the above example).

To ease the transition to the new multiple release scheme, the **`-p`** option allows the system administrator set set the default release for the **`cc`**, **`hc`**, **`cc++`**, and **`c++`** commands independently of **`ec`**, **`ec++`**, and the **`c.*`** utilities. Typically this will be used to set them to use the pre-5.1 releases:

```
c.install -rel pre5.1 -p
```

Note a subsequent invocation of **`c.install`** with the **`-d`** option will override this setting. This is because it is expected that normally the **`cc`**, **`hc`**, **`cc++`**, and **`c++`** commands will invoke the latest release.

The **`-arch`** and **`-osversion`** options set the default architecture and PowerMAX OS versions being targeted. This determines which header files are used in compilation and which libraries are used in linking. A default must be set for the PLDE. Native compilers use the native headers and libraries by default. A future release of PowerMAX OS could ship with cross development support, enabling the use of the **`-arch`** and **`-osversion`** options. The user can override the system defaults by specifying compile options, specifying link options, using **`c.release`**, or setting the environment variables `PDE_ARCH` and `PDE_OSVERSION`.

The **`-target`** option specifies the default target microprocessor. The controls microprocessor specific features (such as AltiVec enhancements), what instructions and intrinsics are available, and how instructions are scheduled. The special target **`ppc`** is guarenteed to generate code compatible with all PowerPC processrs: it restricts code generation to use instructions that are common to all PowerPC implementations. The default may be overidden by the user by specifying the **`-target`** compile option, using **`c.release`**, or setting the environment variable `PDE_TARGET`.

c.instantiation

Manipulate instantiation of templates and extern inlines.

The syntax of the **`c.instantiation`** command is:

`c.instantiation [`*options***`] [`***unit-id***`]`**

The following represents the **`c.instantiation`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-F`	force	Force unit-id to exist. Used to setup template resolutions before any files are compiled, usually via the **`c.script`** command.
`-f`	force	If the unit in the -r option is not in the potential associated units list, force it to be added anyway (otherwise an error is generated)
`-H`	help	Display syntax and options for this function
`-hide` "*unit-list*"	hide	Hide specified normal units from being considered for resolving instantiation requests
`-!hide` "*unit-list*"	unhide	Unhide specified normal units so they may be considered for resolving instantiation requests
`-l`	list	List the potential and actual associated units of unit-id
`-magnet` "*unit-list*"	magnet	Designate specified units as magnets. This means that instantiation automation will prefer them over other units for hosting artificial units for template and extern inline instantiation.
`-!magnet` "*unit-list*"	unmagnet	Remove magnet designation. Specified units will be consider for resolving instantiation only if no magnetic unit can perform the resolution.
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-resolve` "*unit-list*"	resolve	Add specified units to the list of units that will host the instantiation of *unit-id*.
`-!resolve` "*unit-list*"	unresolve	Remove specified units from the list of units that will host the instantiation of *unit-id*.
`-v`	verbose	Display a message for each selected or hidden definition

unit-list is a comma or white space list of *unit-id*s. The list must be enclosed in quotes if more than one unit is specified.

unit-id is defined by the following syntax:

unit

See “Unit Identification” on page 3-4 for more information about the *unit-id*.

c.intro

Introduce source files (and units therein) to the environment

The syntax of the **`c.intro`** command is:

`c.intro [`*options***`] [`***source_file*** `...]`**

The following represents the **`c.intro`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-language [C|C++]`	language	Specify whether the source file is written in **`C`** or **`C++`**
`-make "`*command***`"`**	make command	Specify a command for **`c.build`** to run before checking the timestamp on the introduced source file. Typically, this command will be a "**`make`** *source_file*" command to create a machine-generated source file.
`-o` *path*	output	Specify the path to the object file where the unit will be compiled to. Normally this is **`.c++/.units/`***unitname***`.o`**. This option is intended for internal use. Users should generally avoid it.
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-s` *file_list*	file list	Read *file_list* for a list of files to process If - is specified, read file list from **`stdin`**
`-unit` *name*	unit name	Override the default unit name for the source file. This allows the same source file to be introduced multiple times (to be built in more than one way), or to introduce two source files that share the same basename, but have different paths.
`-v`	verbose	Echo files as they are processed

The **`-s`** option takes as its argument a *file_list* containing the names of all the files to be processed by **`c.intro`**. This is useful in order to introduce many files at once. Each file must be on a separate line in the *file_list.*

If **`-`** is specified for *file_list*, **`c.intro`** uses input from **`stdin`**. This is provided mainly so that users can pipe output from another UNIX command to **`c.intro`**.

Since the unit name **`all`** is reserved to mean all units in various commands, if a file called **`all.c`** is introduced, its default unit name is **`_all`**, and if a file called **`_all.c`** is introduced, its default unit name is **`__all`**, etc.

To reference a foreign unit that hasn't actually be introduced into a foreign environment on the environment search path yet, use "**`c.intro -name`** *name*" without specifying a source file. This declares the name to the local environment so that you can refer to the name without getting an error message. This isn't necessary if the foreign unit has already been introduced into the foreign environment and that environment is on the environment search path.

`c.rmsrc` can be used to eliminate the association of source files with the environment. **`c.rmsrc`** removes all knowledge of source files (and units therein) from the environment. See “c.rmsrc” on page 4-65 for more information.

c.invalid

Force a unit to be inconsistent thus requiring it to be recompiled

The syntax of the **`c.invalid`** command is:

`c.invalid [`*options***`] [`***unit-id*** `...]`**

The following represents the **`c.invalid`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-source` *file*	source file	Invalidate all units built from the specified file
`-t`	template	Erase template instantiation associations
`-v`	verbose	Display a message for each invalidated unit

unit-id is defined by the following syntax:

unit **`| all`**

The **`c.invalid`** tool is used to force a unit to be considered inconsistent, usually to force them to be rebuilt by **`c.build`**. If the **`-t`** option is also specified, template instantiation automation will forget the associates it made between the specified units and any instantiatable entities (such as templates or extern inlines) that it decided to build using this unit's source, but not those made explicitly by the user. See "Template Instantiation" on page 6-3 and "c.instantiation" on page 4-31.

The **`c.touch`** tool is provided to allow the opposite functionality. See "c.touch" on page 4-69.

c.link

Link a partition (an executable, or shared object file)

INTERNAL UTILITY

This tool is used internally by **`c.build`** which is the recommended utility for compilation and program generation.

`c.link` is not intended for general usage.

The syntax of the **`c.link`** command is:

`c.link [`*options***`] [`***link-options***`]`** *partitions* **`...`**

The following represents the **`c.link`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-i`	information	Suppress information messages
`-o` *file*	output	Override the default output for the partition and place the output in *file*
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-V`	verify	Display the link commands, suppressing execution
`-v`	verbose	Display links as they are done
`-vv`	very verbose	Display the link commands before execution
`-w`	warnings	Suppress warning messages

See “Link Options” on page 4-70 for list of link options.

c.ls

List information about the environment

The syntax of the **`c.ls`** command is:

`c.ls [`*options***`]`**

The following represents the **`c.ls`** options:

Option	Meaning	Function
`-A`	all	Dump information about everything information can be dumped: sets **`-E`**, **`-S`**, **`-U`**, **`-T`**, and **`-P`**.
`-all`	all	Include information from all environments on the Environment Search Path.
`-art`	artificial	Include artificial units (those created by the environment to support templates and extern inlines)
`-E`	environment	List attributes of the local environment
`-e`	everything	Provide an all-encompassing listing using the same format as the **`-l`** and **`-v`** options, but omitting nothing that may be known. Often generates an overwhelming amount of output.
`-env` *env*	environment	Specify an environment pathname
`-format` *fmt*	format	Format the information supplied for each unit based on the format descriptor *fmt.* This option may not be used with any option that displays information about the environment other than units.
`-format help`	format help	Display list of format descriptors
`-H`	help	Display syntax and options for this function
`-h`	headers	Suppress headers on long and verbose listings
`-instantiation`	instantiation	Display instantiation information
`-l`	long	List the same information as the **`-1`** option, but use a long format.
`-local`	local	Filter candidate units and partitions to include only those found in the local environment (default)
`-N`	name	Sort lists by name in ascending order
`-n`	number	Include a total count of the number of units, partitions, etc.
`-P`	partition	List information about all partitions. Use **`-all`** or **`-local`** to include partitions in all environments on the environment search path or restrict to local partitions only.
`-plist` "*list*"	partition	List information about specific partitions. *list* is a list of partitions separated by commas or spaces. It is a single parameter, and so must be enclosed in double-quotes if more than one partition is specified. Multiple **`-plist`** options may be specified on the command line.

Option	Meaning	Function
-pfile *filename*	partition	List information about specific partitions. The list of partitions is in the specified filename. The partition names can be one per line, or multiple partitions can be specified on a line, separated by commas and/or spaces. Multiple **-pfile** options may be specified on the command line.
-r	reverse	Reverse the sorting order
-rel *release*	release	Specify a Concurrent C/C++ release (other than the default release)
-S	source	List information about all source files. Use **-all** or **-local** to include source files in all environments on the environment search path or restrict to local source files only.
-shadow	shadow units and partitions	When the **-all** option is not specified, **c.ls** does not scan the foreign environments in the environment search path for all visible units and partitions. The local environment does however contain some information about those units and partitions if they are referenced locally. These "shadows" of the foreign units and partitions are not listed unless explicitly asked for with the **-shadow** option.
-slist *"list"*	source	List information about specific source files. *list* is a list of source files separated by commas or spaces. It is a single parameter, and so must be enclosed in double-quotes if more than one source file is specified. Multiple **-slist** options may be specified on the command line.
-sfile *filename*	source	List information about specific source files. The list of source files is in the specified filename. The source file names can be one per line, or multiple source files can be specified on a line, separated by commas and/or spaces. Multiple **-sfile** options may be specified on the command line.
-t	time	Sort lists by timestamp.
-T	unit	List information about all symbols.
-templates	templates	Show units associated with template instantiation (same as **-art**).
-tlist *"list"*	unit	List information about specific symbols. *list* is a list of symbols separated by commas or spaces. It is a single parameter, and so must be enclosed in double-quotes if more than one symbol is specified. Multiple **-tlist** options may be specified on the command line.
-tfile *filename*	unit	List information about specific symbols. The list of symbols is in the specified filename. The symbol names can be one per line, or multiple symbols can be specified on a line, separated by commas and/or spaces. Multiple **-tfile** options may be specified on the command line.
-U	unit	List information about all units. Use **-all** or **-local** to include units in all environments on the environment search path or restrict to local units only.
-ulist *"list"*	unit	List information about specific units. *list* is a list of units separated by commas or spaces. It is a single parameter, and so must be enclosed in double-quotes if more than one unit is specified. Multiple **-ulist** options may be specified on the command line.

Option	Meaning	Function
-ufile *filename*	unit	List information about specific units. The list of units is in the specified filename. The unit names can be one per line, or multiple units can be specified on a line, separated by commas and/or spaces. Multiple **-ufile** options may be specified on the command line.
-v	verbose	Provide a verbose listing, using the same format as the **-l** option, but providing more information. Empty values are not displayed.
-1	one, single	Display lists with one line per item

unit-id is defined by the following syntax:

unit **| all**

The behavior of **c.ls** with no options or *unit-id* specified is to display only some basic information about the local environments, list the names of all the units within the local environment (if no options are specified, **-local** is assumed), and the names of partitions within the local environment.

To see more information than is provided in a default listing, **c.ls** provides a number of options:

-1	Provides a brief listing of units, partititions, etc, with one item per row and some additional information in multiple columns.
-l	Provides a long listing consisting the same information as the **-1** option.
-v	Provides a verbose listing.
-e	Provides an all-encompassing listing.
-instantiation	Provides a listing of just instantiation information. This option can only be used to display information about units.
-format	Provides a method to display only the fields that are desired. This option can only be used to display information about units.

The options **-l**, **-v**, **-e**, **-format**, and **-1** options are mutually exclusive.

When displaying instantiation information, instantiations set explicitly by the user with the **c.instantiation -resolve** command are marked with "*****" (see "c.instantiation" on page 4-31). Instantiations set explicitly by the user with **#pragma instantiation** or by command line options are marked with "**#**".

Formatting the listing

The **`-format`** option to **`c.ls`** allows you to format the information listed for each unit based on a format descriptor, *fmt*, which takes the form:

```
“%[Modifier]Descriptor random_text %[Modifier]Descriptor...” ...
```

Characters encountered in the quoted format string which are not part of a descriptor are echoed in the output. Any character other than ‘a’..’z’ and ‘_’ serve to terminate the current descriptor; any such characters are echoed.

The descriptors and their potential modifiers are shown below:

Descriptor	Modifier	Meaning
`consistent`	**`CY`**	Is the unit up-to-date with the source: `consistent` or `not consistent`, or `yes` or `no` respectively if **`Y`** modifier is given.
`date`	**`C`**	Timestamp of the object file, or `object file missing` if the unit has never been built.
`environment`	**`CL`**	The native environment of foreign and fetched units. Empty string if the unit is local.
`hidden`	**`CY`**	Is the unit hidden from being considered for hosting an instantiation: `hidden` or `not hidden`, or `yes` or `no` respectively if **`Y`** modifier is given.
`incdate`	**`C`**	Timestamp of the most recently modified included file (including the source file itself), or `include file missing` is the source file or one of the include files is missing.
`kind`	**`C`**	Kind of unit: `normal` or `artificial` (a unit created by the environment for templates and extern inlines).
`language`	**`C`**	Language of the Unit: `C` or `C++`.
`magnet`	**`CY`**	Is the unit prefered over other units for associating artificial units for instantiation automation: `magnet` or `not magnet`, or `yes` or `no` respectively if **`Y`** modifier is used.
`missing`	**`CY`**	Is the unit's source file missing: `missing` or `not missing`, or `yes` or `no` respectively if **`Y`** modifier is used.
`name`	**`CL`**	Name of the unit
`options`	**`PTEQ`**	The **`P`**ermanent, **`T`**emporary, or **`E`**ffective options of the unit, depending on the modifier.
`scrdate`	**`C`**	Timestamp of the source file (not including any include files, or `source file missing` if the source file doesn't exist.
`srcfile`	**`CL`**	Name of the source file
`state`	**`C`**	Is the unit compiled: `compiled` or `not compiled`.
`visa`	**`C`**	Visa of a unit: `native`, `fetched`, `naturalized`, `visiting`, or `foreign`.

The modifiers have the following meanings:

Modifier	**Meaning**	**Description**
C	column	Causes the current item to be padded with sufficient trailing blanks to form a column; this modifier is allowed for any descriptor
L	long	Causes the long-form of the item to be output: date descriptors will include microseconds; path descriptors will be forced into fully-rooted filename notation
Y	yes	Output `yes` or `no`, instead of *X* or `not` *X* respectively.
Q	quote	Quote or escape special shell characters.
E, P, T	options	Selects between the effective, permanent, or temporary option sets; only legal for the option descriptor

For example, in an environment that contains the unit `hello`, the following **`-format`** option to **`c.ls`** produces the following output:

```
$ c.ls -format “%name was built on %date\n”
hello was introduced on Mon Dec  6 15:49:58 1999
```

Sorting

There are a few options to **`c.ls`** with which to sort the output. They are:

-N	Sort by name in ascending order
-t	Sort by timestamp
-r	Reverse the sorting order

c.lssrc

List source files associated with the environment

The syntax of the **`c.lssrc`** command is:

`c.lssrc [`*options***`] [`***source-file***`]`**

The following represents the **`c.lssrc`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-s` *file_list*	file_list	Read *file_list* for a list of files
`-vv`	very verbose	Display command lines used to invoke subordinate tools

`c.lssrc` provides information about source files introduced to the environment. The information available via this tool is specific only to the source file. For information about units contained within the source file, the **`c.ls`** tool should be used. See “c.ls” on page 4-39 for more information.

With no options, **`c.lssrc`** provides a list of the names of all source files introduced to the environment. If a *source-file* name is specified on the command line or the **`-s`** option is used with a file containing a list of source file names, only the mentioned source files will be listed.

c.make

Create a make file for building the partitions of an environment

The syntax of the **`c.make`** command is:

`c.make [`*options***`]`**

The following represents the **`c.make`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-ii`	ii file	Create *filename*.ii targets for template instantiation.
`-no_include`	#include	Omit `#include` files from the units' dependency lists
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-usr_include`	/usr/include	Include system `#include` files from the units' dependency lists.

`c.make` writes a makefile to **`stdout`** for building the partitions local to a particular environment. This is useful for porting to systems not targeted by Concurrent's Program Development Environment.

The generated makefile creates several symbols at the head to assist the user in porting the makefile to other systems:

`EC`	The C compiler to use. Defaults to **`/usr/ccs/bin/ec`**
`CFLAGS`	Options to pass to the C compiler. Empty by default. Each individual compile has its effective options in its own target.
`ECPP`	The C++ compiler to use. Defaults to **`/usr/ccs/bin/ec++`**
`CPPFLAGS`	Options to pass to the C++ compiler. Empty by default.
`OBJ`	The directory where unit's object files will be placed. Defaults to **`.c++/.units`** by default.

The target `all` is output next. This target builds all the partitions.

The targets for the units are output next. If the unit has been compiled in the environment, then the full list of `#include` files are known. These are output into the dependency list. The `#include` files may be omitted from the dependency list with the **`-no_include`** option. Alternatively, system `#include` files (those whose path starts with **`/usr/`**), may be included in the dependency list with the **`-usr_include`** option.

If automatic instantiation is used and instantiations have been assigned to a unit in the environment, either manually or by the environments automatic mechanism, then the make target for a unit can also set up a **`.ii`** file so that the the same instantiations are assigned to each compilation using the non-PDE instantiation automation using the **`-ii`** option. There are several caveats with this though:

- **`.ii`** files use “mangled” names. Thus they may not be compatible between different versions of the compiler.
- Other vendors’ compilers will not use the same mechanism (unless they are based on the same version of the Edison Design Group’s front end and haven’t made changes to name mangling themselves).
- Changes to source may result in changes to the instantiation assignments. The non-PDE instantiation automation is not as flexibile as PDE automation, so as further recompilations alter the **`.ii`** files, some templates (or other instantiatable entities) may not longer get instantiated properly.

Targets for each local partition are output last.

c.man

Invoke/position interactive help system (requires an X™ terminal)

The syntax of the **c.man** command is:

c.man [*options***] [** *manual* **[***topic***]]**

The following represents the **c.man** options:

Option	Meaning	Function
-display *disp*	X display	Select an X terminal
-env *env*	environment	Specify an environment pathname
-l	list	Lists available online manuals
-man *manpage*	man page	Display **man** page for specified *manpage*
-rel *release*	release	Specify a Concurrent C/C++ release (other than the default release)
-H	help	Display syntax and options for this function

c.man invokes the interactive HyperHelp™ system as directed by options and arguments. If a HyperHelp session for the user is already active, **c.man** will position the existing session to the specified topic or manual.

To see a list of the names of each online *manual* available for viewing with HyperHelp, issue:

```
$ c.man -l
```

To open a specific *manual*, issue **c.man** with the name of that *manual*:

```
$ c.man c++.LAG
```

NOTE

The manual **c++.LAG** points to the latest-and-greatest version of the *Concurrent C/C++ Reference Manual*.

If the manual is not recognized (and is not interpreted as a *topic*), then HyperHelp is opened to the Bookshelf.

To view a particular *topic* within a specific *manual*, issue either that *topic* along with the *manual* in which it is contained, or the *topic* alone.

```
$ c.man c++.LAG c.build
```

or

```
$ c.man c.build
```

will position the HyperHelp system to the description of the **`c.build`** command.

Topics for the C/C++ Reference Manual include the names of all Concurrent C/C++ utilities, all pragmas recognized by Concurrent C/C++, and various C/C^{++} bindings.

NOTE

The *topic* argument is meant as a shortcut for positioning the HyperHelp session. The list of topics recognized by **`c.man`** is short and obviously not meant to be comprehensive. Direct use of HyperHelp is intended for general manual browsing and selection.

If a *topic* is not recognized, but the *manual* is, HyperHelp will be positioned at the `Find` window for that *manual*.

c.mkenv

Create an environment which is required for compilation, linking, etc.

The syntax of the **c.mkenv** command is:

c.mkenv [*options***] [--] [***compile_options***] [***environment_pathname***]**

The following represents the **c.mkenv** options:

Option	Meaning	Function
-arch *arch*	architecture	Add **-arch** to both the environment-wide default link options and the environment-wide default compile options.
-env *env*	environment	Specify an environment pathname
-f	force	Force environment creation even if it or some portion of it already exists
-H	help	Display syntax and options for this function
-language *lang*	language	Select **C** or **C++** as language (*lang*) to be used as the default in **c.intro** invocations.
-oset *"options"*	link options	Set the environment-wide default link options.
-osversion *osversion*	OS version	Add **-osversion** to both the environment-wide default link options and the environment-wide default compile options.
-rel *release*	release	Specify a Concurrent C/C++ release (other than the default release)
-src *"dir* **[***options***]** **%f"**	source management	Run pre- and post-edit scripts for interfacing with source management whenever file is edited with **c.edit**.

The *compile_options* specified with this command become the permanent environment-wide compile options and apply to all units introduced into this environment. They may be changed by using **c.options**. They may also be overridden by temporary environment-wide compile options, or for particular units by permanent or temporary unit options or pragmas. See "Compile Options" on page 3-9 for a more detailed explanation of this relationship.

The **-f** option forces creation of an environment even if one has already been created or if only a portion of it already exists. If the **c.mkenv** tool is interrupted or fails for some reason (such as not enough disk space, power failure, etc.), the creation of the environment may not have completed. Trying to recover from this failure by running the **c.mkenv** tool again may result in a message similar to the following:

```
c.mkenv: database file .C++ already exists
          in environment /some_dir/env_dir.
```

The **-f** option will force this environment to be created, thereby overriding such error messages.

Use **`c.options -HC`** for a list of *compile_options*. Also, “Link Options” on page 4-70 provides a similar list.

An environment can be removed with **`c.rmenv`**. See “c.rmenv” on page 4-64 for details.

c.options

Set compilation options for units or the environment

The syntax of the **c.options** command is:

c.options [*options***] [--] [***compile_options***] [***unit-id* **...]**

The following represents the **c.options** options:

Option	Meaning	Function
-clear	clear	Clear all designated options for the specified entities
-default	default	Operate on the default options for the entire environment
-del	delete	Delete the designated options from the specified entities
-eff	effective	Display the effective options (based on temporary, permanent, environment defaults)
-env *env*	environment	Specify an environment pathname
-H	help	Display syntax and options for this function
-h	header	Remove the header from the option list output
-HC	help compile	Display list of compile options
-HQ	help qualifier	Display list of qualifier keywords (**-Q** options)
-keeptemp	keep temporaries	Propagate the temporary options for the units into the set of permanent options
-language *lang*	language	Select **C** or **C++** as language to be used
-list	list	List the option sets for the specified entities
-make "*command***"**	make	Change the make command associated with the primary source file of the specified units.
-mod	modify	Modify the designated options for the specified entities
-perm	permanent	Operate on the permanent options (this is the default)
-rel *release*	release	Specify a Concurrent C/C++ release (other than the default release)
-set	set	Set the designated options for the specified entities
-source *file*	source file	Change options on units built from specified primary source file. Most commonly used with **-make** option.
-src "*dir* **[***options***] %f"**	source management	Specify pre- and post-edit scripts for interfacing with source management when editing source files with **c.edit**.
-temp	temporary	Operate on the temporary options
-v	verbose	Display a message for each change
--	separator	Separate options to **c.options** from compiler options. Optional if there is no ambiguity.

unit-id is defined by the following syntax:

unit **| all**

Option Sets

As discussed in “Compile Options” on page 3-9, there are different levels of options in Concurrent C/C++. These option sets are designated by the following flags to **`c.options`**:

Flag	Designation	Operates on
`-default -perm`	environment-wide compile options	all units
`-default -temp`	temporary environment-wide compile options.	all units
`-perm`	permanent compile options	specified units
`-temp`	temporary compile options	specified units

In addition, the *effective options* are derived from these four and their hierarchical relationship to one another. This set is discussed in greater detail in “Effective Options” on page 3-11.

Listing options

The option sets may be viewed using the **`-list`** option to **`c.options`**. When issued alone, **`-list`** shows the permanent, temporary, and effective option sets for the units specified. For example, the following command lists those option sets for the unit `hello`,

```
$ c.options -list hello
```

By combining the **`-list`** option and the desired option set’s flag, only that option set is displayed for the specified units. For instance, to view the permanent options for the unit `hello`,

```
$ c.options -list -perm hello
```

This only lists the permanent options for the units specified. You may specify multiple unit names, or you may use the keyword **`all`** to specify all units in the environment.

To list the effective options for all units in the environment,

```
$ c.options -list -eff all
```

However, this particular option does the same thing when issued alone,

```
$ c.options -eff all
```

Note that since the **`-default`** flag operates on all the units in the environment by definition, there is no need to specify any unit names. To list the default options,

```
$ c.options -list -default
```

Setting options

The option sets may be initialized or reset by using the **`-set`** flag to **`c.options`**. This sets the specified options for the units designated. Any previous options for the set designated are replaced. For example,

```
$ c.options -set -perm -g hello
```

turns on debug information in the permanent option set for the unit `hello`.

If the following command is issued,

```
$ c.options -set -perm -O1 hello
```

the permanent option set will only contain the **`-O1`** option (the previous **`-g`** option will have been replaced).

Modifying options

In order to modify an option set, the **`-mod`** flag is used with **`c.options`**. This flag adds the specified options to the designated set, while retaining any other options that existed in this grouping. For instance, after the following command,

```
$ c.options -set -temp -g hello
```

the temporary option set for the unit `hello` consists of **`-g`**.

To add an optimization compile option to this set,

```
$ c.options -mod -temp -O2 hello
```

The temporary option set for `hello` now consists of **`-g`** and **`-O2`**.

Clearing options

All of the options may be cleared from a designated option set by using the **`-clear`** option to **`c.options`**. To clear all of the temporary options from all units in the environment,

```
$ c.options -clear -temp all
```

Deleting options

Using the **`-del`** flag with **`c.options`** is more specific than using the **`-clear`** option and allows specified options to be deleted from a particular option set.

For example, if the environment-wide compile option set (**`-default`**) contains **`-O2`**, **`-!g`** and **`-S`**, the following command,

```
$ c.options -del -!g -default
```

will remove the **`-!g`** option from the set and leave **`-O2`** and **`-S`** to remain as the environment-wide compile options.

Some sets of options are mutually exclusive because they effectively set an attribute to a particular value. For example, **`--early_tiebreaker`** and **`--late_tiebreaker`**. Specifying one will remove the other from the effective option list. Similarly, specifying **`--!early_tiebreaker`** will actually remove any tiebreaker setting from the effective options list.

Keeping temporary options

Temporary options may be propagated into the permanent set by using the **`-keeptemp`** option to **`c.options`**. This moves the temporary options into the permanent option set and clears the temporary set. The following command does this for all units in the environment,

```
$ c.options -keeptemp all
```

See “Link Options” on page 4-70 for more information.

Also, see the example of this in “What are my options?” on page 2-6.

Setting options on foreign units

Options for units in foreign environments cannot be changed using **`c.options`** in the local environment. In order to change the options on a foreign unit, it must first be fetched (see “c.fetch” on page 4-24).

c.partition

Define or display a partition for the linker

The syntax of the **`c.partition`** command is:

`c.partition [`*options***`] [`***partitions* **`...]`**

The following represents the **`c.partition`** options:

Option	Meaning	Function
`-a`	all	Display all partitions in the environment (Normally, only those originating in the environment are displayed)
`-add "`*units***`"`**	add	Add *units* to the partitions while retaining previously added units *units* is a single parameter; the names of individual units should be comma-separated and enclosed in double quotes
`-addfile` *file*	add from file	As **`-add`**, but reads units from *file*
`-create` *kind*	create	Create the new named partitions as *kind* where *kind* could be **`executable`** (**`exe`**), **`shared_object`** (**`so`**), **`archive`** (**`ar`**), or **`object`** (**`obj`**). See also discussion of `shadow` below.
`-default`	default link options	Direct **`-oappend`**, **`-oprepend`**, and **`-oset`** to operate on the environment-wide link options.
`-del "`*units***`"`**	delete	Delete *units* from the partitions *units* is a single parameter; the names of individual units should be comma-separated and enclosed in double quotes
`-delfile` *file*	delete from file	As **`-del`**, but reads units from *file*
`-env` *env*	environment	Specify an environment pathname
`-f`	force	Force creation of existing partitions and removal of nonexistent partitions
`-H`	help	Display syntax and options for this function
`-HL`	help link	Display link options
`-List`	list all	Display all partitions and information about them
`-list`	list	List all partition names
`-o` *file*	output	Set the name of the corresponding partition output file to be created
`-oappend` *opts*	append link options	Appends the *opts* argument to the link option listing *opts* is a single parameter; it must be enclosed in double quotes
`-oclear`	clear link options	Clear the link options

Option	Meaning	Function
`-oprepend` *opts*	prepend link options	Prepends the *opts* argument to the link option listing *opts* is a single parameter; it must be enclosed in double quotes
`-oset` *opts*	set link options	Set the link options as indicated by *opts* *opts* is a single parameter; it must be enclosed in double quotes
`-parts` *list*	partition list	Set the dependent (comma-separated) partition list for each partition
`-refs`	references	Remove local references to a partition when it is removed. Otherwise, the environment will attempt to find a foreign partition to satisfy any partition dependent on the removed partition.
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-remove`	remove	Remove the specified partitions
`-set "`*units***`"`**	set	Add *units* to the partitions, and remove all others *units* is a single parameter; the names of individual units should be comma-separated and enclosed in double quotes (see below)
`-setfile` *file*	set from file	As **`-set`**, but reads units from *file*
`-v`	verbose	verbose
`-vv`	veri verbose	Display command lines invoking subordinate tools

units is defined by the following syntax:

`[[`*unit_name***`[!][,[+-]`***unit_name***`[!]]...`** (comma-separated list)

+	indicates an included unit (the default)
-	indicates an excluded unit
!	indicates all units directly or indirectly required by the given unit
`all`	as the *unit_name* is special. It means all units, not at the time the command is issued, but at the time **`c.prelink`** is done. **`c.ls`** will show all as a included unit before all explicitly specified units if it is present.

You may specify multiple *partitions* to **`c.partition`** and all *options* specified will apply to every one of those *partitions*. Each *option*, however, may only be specified once. If a particular *option* is repeated on the command line, the last occurrence of that *option* overrides all others.

`!` only can pull in native and fetched units. Use of **`!`** forces all units in the environment to get built because their symbol tables must be used to determine dependencies.

The idiom "**`+unit!,-unit`**" means to include all units that **`unit`** directly or indirectly requires, but not include **`unit`** itself.

If no action is requested, then **`c.partition`** will list information about the named partitions (the equivalent of **`c.ls -v -plist`**).

To reference a foreign partition before the foreign environment has defined it or before the foreign environment as been added to the environment search path, it is necessary to declare it to the local environment with the "**`c.partition -create shadow`** *name*" command. Otherwise, referencing an unknown partition name results in an error.

Link Options

Link options are specified for a particular partition by using the following options to **`c.partition`**:

`-oset` *opts*	Sets the link options as indicated by *opts*
`-oappend` *opts*	Appends the *opts* argument to the link option listing
`-oprepend` *opts*	Prepends the *opts* argument to the link option listing
`-oclear`	Clears the link options

opts is a single parameter containing one or more link options; it must be enclosed in double quotes.

NOTE

Be sure to specify the link options within the double quotes and ensure that they are specified as listed on page 4-70. For example, if the link option **`-bound`** is desired, the leading "**`-`**" must be specified as well.

For example, to set the link options for the partition **`hello`** to include the link options **`--strip`** and **`-udump`**:

```
$  c.partition -oset "--strip -udump" hello
```

Issuing **`c.ls`** will show the link options for this partition:

```
$  c.ls -plist hello -v
   PARTITION: hello
      nationality          : native
      kind                 : executable
      output file          : hello
      link options         : --strip -udump
      included units (+)   :
      hello
```

To append a link option to this set, use the **`-oappend`** option:

```
$  c.partition -oappend "-lm" hello
```

The link options now will be:

```
$  c.ls -plist hello -v
   PARTITION: hello
      nationality          : native
      kind                 : executable
      output file          : hello
      link options         : --strip -udump -lm
      included units (+)   :
      hello
```

To clear all link options for this partition, use the **`-oclear`** option:

```
$  c.partition -oclear hello
```

See “Link Options” on page 4-70 for a list of link options.

Also, “Link Options” on page 3-8 provides further discussion of this topic.

If the **`-default`** option is specified, **`-oset`**, **`-oprepend`**, and **`-oappend`** also manipulate the enviroment-wide default link options. These options are implicitly prepended to an individual partition’s link options. Only those link options appropriate to the type of partition are so prepended.

c.path

Display or change the Environment Search Path for an environment

The syntax of the **`c.path`** command is:

`c.path [`*options***`]`**

The following represents the **`c.path`** options:

Option	Meaning	Function
`-A` *path*	append	Append *path* to the end of the Environment Search Path
`-a` *path1* **`[`***path2***`]`**	append	Append *path1* after *path2*. If *path2* is not specified, this option is identical to the **`-A`** option
`-env` *env*	environment	Specify an environment pathname
`-f`	full path	Display full environment pathnames
`-H`	help	Display syntax and options for this function
`-I` *path*	insert	Insert *path* at the beginning of the Environment Search Path
`-i` *path1* **`[`***path2***`]`**	insert	Insert *path1* before *path2*. If *path2* is not specified, this option is identical to the **`-I`** option
`-P`	purge	Remove all paths in the Environment Search Path
`-R` *path1 path2*	replace	Replace *path1* with *path2*
`-r` *path*	remove	Remove *path* from the Environment Search Path
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-t`	transitive	Display transitive closure of environments in the Environment Search Path
`-v`	verbose	If combined with any other **`c.path`** option, display the Environment Search Path after the operation is complete
`-w`	warnings	Suppress warning messages
`-x` *path*	exclude	Remove all but *path* from the Environment Search Path

Concurrent C/C++ uses the concept of an Environment Search Path to allow users to specify that units and partitions from environments other than the current environment should be made available in the current environment. See “Environment Search Path” on page 3-2 for a more complete discussion.

c.prelink

Resolve transitive closure of included units and template instantiation before linking.

INTERNAL UTILITY

This tool is used internally by **`c.build`** which is the recommended utility for compilation and program generation.

`c.prelink` is not intended for general usage.

The syntax of the **`c.prelink`** command is:

`c.prelink [`*options***`] [`***partitions***`]`**

The following represents the **`c.prelink`** options:

Option	**Meaning**	**Function**
`-auto_instantiation`	automatically instantiate	Instantiate any templates used by units in the partition (this is the default for executable partitions)
`-C "`*compiler***`"`**	compiler	Specify alternate compiler when compiling template instantiations
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-noimport`	no import	Suppress naturalization to resolve templates
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-ufile` *file*	units from file	Prelink a list of units as though they were a partition
`-ulist` "*list*"	units from list	Prelink a list of units as though they were a partition
`-v`	verbose	Print header for each compilation
`-vv`	very verbose	Print subordinate tool command lines

This tool takes care of template instantiation automation, selection of units that are going to be included in the final link, and other bookkeeping activities that must be performed before actually linking a partition.

c.release

Display release installation information

The syntax of the **`c.release`** command is:

`c.release [`*options***`]`**

The following represents the **`c.release`** options:

Option	Meaning	Function
`-arch` *arch*	default arch	Set the user's default architecture target (nh, mot, synergy, etc.)
`-e`	env	Display the path of the selected environment
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-n`	name	Display the name of the selected release
`-osversion` *osversion*	default os ver.	Set the user's default osversion target
`-p`	path	Display the path to the selected release
`-q`	query	Display the selected environment and release
`-r`	remove	Remove the default release currently set for the invoking user
`-rel` *release*	release	Specify a Concurrent C/C++ release
`-S`	system default release	Specify system default release (ignoring user default, PDE_RELEASE environment variable, etc.)
`-target`	default target	Set user's default target microprocessor
`-U`	user default release	Specify the user default release (ignoring system default, PDE RELEASE environment variable, etc.)
`-u`	user	Set the default release for the invoking user

If invoked without options, **`c.release`** lists all available release installations on the current host. For example,

```
$  c.release
```

provides output similar to the following (the native compiler will not output OS Version and Architecture information since only the PLDE currently supports cross compilation to multiple versions of PowerMAX OS and multiple Concurrent supported architectures):

```
The following compiler releases are available on this machine:

   Name              Path
   ----              ----
   5.1               /usr/opt/plde-c++-5.1
   5.2               /usr/opt/plde-c++-5.2
 * 5.3               /usr/opt/plde-c++-5.3

The following cross target OS releases are available on this machine:

   Version           Architecture(s)
   -------           ---------------
 * 4.3               moto, * nh
   5.0               synergy

The default target microprocessor is ppc604e

(*) Designates the default
```

Screen 4-3. c.release output

The **-q** option displays the release for the specified environment (or the local environment if no environment is specified). For example,

```
$  c.release -q
```

in a Concurrent C/C++ environment named **test** provides the following output:

```
environment path: /csteam/vir/home/jgj/test
release name: 5.3
release path: /usr/opt/plde-c++-5.3
```

Screen 4-4. c.release -q output

c.release may be invoked with any combination of **-rel** and/or **-env** options. All remaining options are mutually exclusive, and may not be combined in a single invocation of **c.release**.

c.report

Generate profile reports in conjunction with `c.analyze -P`

The syntax of the **`c.report`** command is:

`c.report [`*options***`]`** *executable_file* **`[`***executable_file***`.prof]`**

The following represents the **`c.report`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-v`	verbose	Print invocations of subprocesses
`-H`	help	Display syntax and options for this function
`-a`	all	List information from all individual runs even if **`-t`** option is on
`-b`	basic	List basic block statistics
`-B` *num*	expensive basic	List only the *num* most expensive basic blocks
`-B` *num%*	% time basic	List only the basic blocks where the first *num***%** of time was spent
`-c`	calls	For each routine, list calls it makes
`-d` *rng*	data range	Restrict range of data sets examined
`-f`	for each	For each routine, list who calls it
`-i`	info	List summary information for the whole run
`-l`	max	Use max time instead of min time of basic block
`-M` *hz*	Mhz	Specify assumed megahertz clock rate for computing wall time
`-m`	milliseconds	Print milliseconds rather than cycles for most reports
`-n`	miss	List data access cache miss statistics
`-N` *num*	data acc miss	List only the *num* most numerous data access secondary cache misses
`-o`	cache miss	List instruction cache miss statistics
`-O` *num*	secondary miss	List only the *num* most numerous instruction secondary cache misses
`-r`	routine	List routine statistics
`-R` *num*	expensive routine	List only the *num* most expensive routines
`-R` *num%*	% time routine	List only the routines that use the first *num***%** of time
`-s`	summary	List header summarizing data set from each run
`-t`	total	Total all data sets and list cumulative times

Option	**Meaning**	**Function**
`-T` *file*	dump	Dump sum of all data sets into specified *file*
`-w`	readable	Just dump the raw profile data in human readable form
`-z`	zero	List routines and basic blocks with zero time

*executable_file.***`prof`** is the name of the profile data file generated by running the program. The default is the program name with the suffix **`.prof`**.

See “c.analyze” on page 4-4 for more information.

c.restore

Restore a corrupted database

The syntax of the **c.restore** command is:

c.restore [*options***]**

The following represents the **c.restore** options:

Option	Meaning	Function
-env *env*	environment	Specify an environment pathname
-H	help	Display syntax and options for this function
-n *n*	number	Select which backup of the database to restore
-rel *release*	release	Specify a Concurrent C/C++ release (other than the default release)
-undo	undo	Undo the last restore

The **c.build** command makes a backup of the database each time it is invoked from the command line (recursive invokations do not trigger a new backiup). Three such backups are maintained, numbered 1, 2, and 3, where 1 is the most recent. Should a power outage crash the computer in middle of updating the database, it might be left in an inconsistant state. The **c.restore** command will restore the database from the specified backup, and remove any unit or partition that is newer than the restored database. This is because the newer units and partitions may be built in a way that is inconsistant with the previous configuration in the database, and so must be rebuilt to insure consistancy.

c.rmenv

Destroy an environment; compilation, linking, etc. no longer possible

The syntax of the **`c.rmenv`** command is:

`c.rmenv [`*options***`]`** *environment_pathname*

The following represents the **`c.rmenv`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-f`	force	Force an environment destruction, even if it or some portion of it does not exist
`-H`	help	Display syntax and options for this function
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)

Removes an environment, including all units, their state information, and any partition definitions. The source files and any built partitions are left intact after this operation.

The **`-f`** option can be used to force an environment's destruction, even if some portion of it does not exist. For example, if the **`c.mkenv`** utility was interrupted during its execution (due to not enough disk space, power failure, etc.), the environment may not have been successfully created. If the environment cannot be recognized as valid, Concurrent C/C++ will fail with a message similar to the following:

```
c.rmenv: database file .c++/.database doesn't
         exist in environment earth.
```

The **`-f`** option will force this environment to be removed, thereby overriding such error messages.

The environment can be re-created with **`c.mkenv`** (see page 4-47), but it will be empty and any state will have to be reconstructed by the user.

c.rmsrc

Remove knowledge of source files (and units therein) from the environment

The syntax of the **`c.rmsrc`** command is:

`c.rmsrc [`*options***`] [`***source_file* **`...]`**

The following represents the **`c.rmsrc`** options:

Option	Meaning	Function
`-all`	remove all	Remove all units in the current environment
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-r`	remove	Remove the actual source files
`-refs`	references	Remove references to removed units. Otherwise, the environment will attempt to satisfy those references by searching the environment search path for a unit of the same name.
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-s` *file*	file list	*file* is assumed to be a file containing list of files. When this option is given, **`c.rmsrc`** reads *file* and removes each file in the list
`-v`	verbose	Display a message for each removed source file
`-vv`	very verbose	Echo removed units to **`stdout`**

The **`c.intro`** tool can be used to re-associate the source files (and units therein) with the environment, but those units will be re-created in the `uncompiled` state (see “Compilation States” on page 3-11).

c.script

Produce a script of c.* commands to reproduce the current environment

The syntax of the **`c.script`** command is:

```
c.script
```

The following represents the **`c.script`** options:

Option	Meaning	Function
`-allparts`	all partitions	Restrict script to recreating partitions; units are not recreated
`-echo`	echo	Include echo commands in generated script to indicate progress
`-env` *env*	environment	Specify an environment pathname
`-executables`	recreate executables	Restrict script to recreating executable partitions; units are not recreated
`-H`	help	Display syntax and options for this function
`-length` *size*	restrict length	Restrict length of command lines output to script
`-no_mkenv`	no mkenv	Restrict script from creating a new environment and setting environment wide options
`-no_shadow`	no shadow	Restrict script from declaring referenced units and partitions that are not being recreated. Only useful with the **`-allparts, -pfile`**, or **`-plist`** options
`-pfile` *file*	recreate partitions in file	Restrict script to recreating those partitions specified in *file*; units are not recreated
`-plist "`*part-list***`"`**	recreate partitions in list	Restrict script to recreating specified partitions; units are not recreated
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-simple`	simple script	Make the generated script "simple", meaning each command invocation is limited to a single unit. The resulting script is much easier to read, and thus manually edit, but will run much much slower.

The **`c.script`** tool can be used to reconstruct an environment at a later time. For example, a future release of the tools may not use a compatible database format, so it will be necessary to use **`c.script`** in the old release, then run the script using the new release's commands.

The **`-no_mkenv`** option suppresses the generatation of commands that create the environment and that set environment-wide options. This would be used to merge all or part of one environment into another, for example.

The **-allparts**, **-plist**, **-pfile**, and **-executables** options restrict the script to just recreating some or all partitions. Referenced units and unspecified partitions are declared so that they can be found on the environment search path, unless the **-no_shadow** option is specified.

Consider the following senario: environment **A** is a huge environment with hundreds of units, dozens of archives, and scores of executable programs. It is frozen. A developer wishes to test a small change to one program without modifying enviornment **A** and without rebuilding everything. The developer could do the following, starting in environment **A**'s directory:

```
$ c.script -plist program_1 > ../B/doit.sh
$ cd ../B
$ sh ./doit.sh
$ c.path -A ../A
```

Now, environment **B** can see all the units and partitions of environment **A** as Foreign units and partitions, except **program_1**, which is Native. Building **program_1** in environment **B** will use the units and libraries from environment **A**.

Now, to actually try the change, the developer could:

```
$ cp ../A/unit_1.c .
$ c.intro unit_1.c
$ c.edit unit_1
$ c.partition -add unit_1 program_1
$ c.build program_1
```

This forces **program_1** to link with environment **B**'s local copy of **unit_1** rather than picking up the un-fixed copy in an archive in environment **A**. Now, let's say that because of link order issues, it turns out that **unit_1** must be picked up form the archive. So now the developer wants to export the definition of the archive in environment **A**, but doesn't need, or want, to re-create the environment or re-declare the other units and partitions. So the developer could:

```
$ cd ../A
$ c.script -no_mkenv -no_shadow \
           -plist archive_1 > ../B/doit.sh
$ cd ../B
$ sh ./doit.sh
$ c.partition -del unit_1 program_1
$ c.build program_1
```

Now when **program_1** is built, the program development environment finds the dependant partition **archive_1** to be a local partition that isn't built yet, so it recursively builds it. All the units that go into **`archive_1`** come from environment **`A`**, except **`unit_1`**, which is built locally.

The script generated by **`c.script`** can take several options itself. These are:

Option	**Meaning**	**Function**
`-env` *env*	environment	Specify an environment pathname
`-f`	force	Force creation of new environment if one is already present. Otherwise, the script exits immediately if the environment already exists.
`-H`	help	Display syntax and options for this function
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)

The informal convention is to use the extender **`.pde`** on filenames that are **`c.script`**-generated scripts. This is what **`NightBench(1)`** uses and looks for by default.

c.touch

Make the environment consider a unit consistent with its source file's timestamp

The syntax of the **`c.touch`** command is:

`c.touch [`*options***`] [`***unit-id ...***`]`**

The following represents the **`c.touch`** options:

Option	Meaning	Function
`-env` *env*	environment	Specify an environment pathname
`-H`	help	Display syntax and options for this function
`-rel` *release*	release	Specify a Concurrent C/C++ release (other than the default release)
`-source` *file*	source file	Touch all units in the specified file
`-v`	verbose	Display a message for each invalidated unit

unit-id is defined by the following syntax:

unit **`| all`**

The **`c.touch`** tool is used to force a unit to be considered consistent with its source file, usually to keep it from being rebuilt by **`c.build`**. Note that it may still be considered inconsistent for other reasons.

The **`c.invalid`** tool is provided to allow the opposite functionality. See "c.invalid" on page 4-34 for details.

Link Options

NOTE

Many link options have more than one spelling.

Option	Function
`--14`	Truncate filenames in archives to 14 characters. Caveat: does not detect name collisions!
`--arch` *arch*	Select which architecture's libraries to link with. Currently only valid for the PLDE.
`--auto_instantiation`	Perform automatic template instantiation
`-c`	Suppress message generated when new archive is created
`--c` **`--c++`**	Force linking as C or C++ program.
`-e`*sym* **`--entry_point=`***sym*	Set the entry point address for the output file to be that of the symbol *sym*.
`-f`	Cause archive to be a "fast updating" archive.
`-h`*name* **`--dynamic_link_name=`***name* **`-sp`***name*	Record *name* in the object's `dynamic` section. *name* will be used by the dynamic linker as the pathname of the shared object to search for at run time.
`-L`*dir* **`--library_directory=`***dir*	Add *dir* to the library search directories. This will effect subsequent **`-l`** link options.
`-l`*lib* **`--library=`***lib*	Search the library search path for a lib*lib*.so or lib*lib*.a file to link with.
`-M`*mapfile* **`--mapfile=`***mapfile*	Read mapfile as a text files of directives to ld. Use of this option is strongly discouraged.
`-m` **`--memory_map`**	Produce a memory map of the input/output sections on stdout.

`-O[`*level***`,[no_]post_linker]`**
`-O[no_]analyze`
`--optimization_level=`*level*
`--post_linker`
`--no_post_linker`
	Invoke the post-link optimization (**`analyze(1)`**), select faster libraries by default, etc. See "Optimization" in *Compilation Systems Volume 2 (Concepts)* for detailed discussion of link-time and post-link-time optimizations.

`-osversion` *osversion*	Select which PowerMAX OS version's libraries to link with. Currently only valid for the PLDE.

Option	Function
-P*partition*	Link with the named partition. This is the same as specifying a partition with the **-parts** option of **c.partition**, except that using the **-P** link option allows the user to control the order that partitions are loaded with respect to each other and to other object files and link options.
-Qfpexcept=*precision* **--fpexcept=***precision*	Initialize the machine state register to indicate the kind of floating-point exceptions that can be taken. *precision* can be **imprecise** (the default), **precise**, or **disabled**.
--no_demangling	Suppress demangling ld error messages
-Qno_vendor_reloc	Do not output relocation information in the vendor section of the object file for use by the **analyze(1)** tool. Post-link optimization will not be possible if this option is used.
-Qreentrant_library	Disallow the implied use of the nonreentrant C library libnc.
-s **--strip**	Strip symbolic information from the output file.
-sl	Create a symbolic link from the output file of the partition to the path specified by the **-sp** link option.
--static_Cruntime	Force a dynamically linked program to link with the static form of the C++ runtime.
-U*unit*	Link with the named unit. This is the same as specifying a unit with the **-add** option of **c.partition**, except using the **-U** link option allows the user to control the order that units are loaded with respect to each other and other link options.
-u*sym* **--undefined_linker_symbol=***sym*	Treat *sym* as an undefined symbol that must be resolved.
-v	Verbose output from **ar(1)** and **ec(1)**.
-V	Verbose output from **ld(1)**.
-Wa,*option* **--pass_to_analyze=***option*	Pass an option directory to **analyze(1)**.
-Wl,*option* **--pass_to_linker=***option*	Pass an option directly to **ld(1)**. Use of this option is strongly discouraged.
-X	Do not look in alternative search paths for libraries.
-x **--reduce_symbols**	Do not preserve local symbols with type STT_NOTYPE.
-Zlibs=*mode* **--library_linkage=***mode*	Govern library inclusion. *mode* may be **dynamic**, to direct subsequent **-l** link options to search for shared objects before trying static libraries, or **static**, to direct subsequent **-l** options to search only for static libraries
-Zlink=*mode* **--link_mode=***mode*	Select whether to link **static** or **dynamic**.

Option	**Function**
`-Zsymbolic` **`--symbolic`**	In shared object, bind references to global symbols to their definitions when in the object, if definitions are available. Normally, references to global symbols within shared objects are not bound until run time, even if definitions are available, so that definitions of the same symbol in an executable or other shared objects can overed the object's own definition.

`-zdefs`
`--linker_z=defs`
	Force a fatal error if any undefined symbols remain at the end of the link. This is the default for executable partitions.

`-zlowzeros`
`-zlowzeroes`
`--linker_z=lowzeros`
`--linker_z=lowzeroes`
	Support dereferencing of NULL pointers. See **`ld(1)`**.

`-znodefs`
`--linker_z=nodefs`
	Allow undefined symbols. This is the default for shared object partitions.

`-ztext`
`--linker_z=text`
	Force a fatal error if any relocations against non-writable, allocatable sections remain.

5
Dialects

5 Dialects

This chapter discusses the various dialects of C++ and C that are supported by the Concurrent C++ compiler. The following topics are covered:

- C++ Dialect Accepted
  - New Language Features Accepted
  - New Language Features Not Accepted
  - Anachronisms Accepted
  - Extensions Accepted in Normal C++ Mode
  - Extensions Accepted in **`cfront`** 2.1 Compatibility Mode
  - Extensions Accepted in **`cfront`** 2.1 and 3.0 Compatibility Mode
- C Dialect Accepted
  - C9X Extensions
  - ANSI C Extensions
  - K&R/pcc Mode
  - Extensions Accepted in SVR4 C Compatibility Mode

See Chapter 7 (“Compilation Modes”) for information on options to select these modes.

C++ Dialect Accepted

The front end accepts the C++ language as defined by the ISO/IEC 14882:1998 standard, with the exceptions listed below.

The front end also has a **`cfront`** compatibility mode, which duplicates a number of “features” and bugs of **`cfront`** 2.1 and 3.0.x. Complete compatibility is not guaranteed or intended—the mode is there to allow programmers who have unwittingly used **`cfront`** features to continue to compile their existing code. In particular, if a program gets an error when compiled by **`cfront`**, the EDG front end may produce a different error or no error at all.

Command-line options are also available to enable and disable anachronisms and strict standard-conformance checking.

New Language Features Accepted

The following features not in traditional C++[1] but in the standard are implemented:

- The dependent statement of an `if`, `while`, `do-while`, or `for` is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.
- The expression tested in an `if`, `while`, `do-while`, or `for`, as the first operand of a "`?`" operator, or as an operand of the "`&&`", "`||`", or "`!`" operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.
- Qualified names are allowed in elaborated type specifiers.
- A global-scope qualifier is allowed in member references of the form `x.::A::B` and `p->::A::B`.
- The precedence of the third operand of the "`?`" operator is changed.
- If control reaches the end of the `main()` routine, and `main()` has an integral return type, it is treated as if a `return 0;` statement were executed.
- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.
- A functional-notation cast of the form `A()` can be used even if `A` is a class without a (nontrivial) constructor. The temporary created gets the same default initialization to zero as a static object of the class type.
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.
- Template friend declarations and definitions are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have nontype template parameters.
- A reference to `const volatile` cannot be bound to an rvalue.
- Qualification conversions such as conversion from `T**` to `T const * const *` are allowed.
- Digraphs are recognized.
- Operator keywords (e.g., `and`, `bitand`, etc.) are recognized.
- Static data member declarations can be used to declare member constants.
- `wchar_t` is recognized as a keyword and a distinct type.
- `bool` is recognized.

1. The C++ language of "*The Annotated C++ Reference Manual*" by Ellis and Stroustrup.

- RTTI (runtime type identification), including `dynamic_cast` and the `typeid` operator, is implemented.
- Declarations in tested conditions (in `if`, `switch`, `for`, and `while` statements) are supported.
- Array `new` and `delete` are implemented.
- New-style casts (`static_cast`, `reinterpret_cast`, and `const_cast`) are implemented.
- Definition of a nested class outside its enclosing class is allowed.
- `mutable` is accepted on nonstatic data member declarations.
- Namespaces are implemented, including `using` declarations and directives. Access declarations are broadened to match the corresponding `using` declarations.
- Explicit instantiation of templates is implemented.
- The `typename` keyword is recognized.
- `explicit` is accepted to declare non-converting constructors.
- The scope of a variable declared in the `for-init-statement` of a `for` loop is the scope of the loop (not the surrounding scope).
- Member templates are implemented.
- The new specialization syntax (using “`template <>`”) is implemented.
- Cv-qualifiers are retained on rvalues (in particular, on function return values).
- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between PODs and non-PODs with trivial constructors.
- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions).
- `extern inline` functions are supported, and the default linkage for `inline` functions is external.
- A typedef name may be used in an explicit destructor call.
- Placement delete is implemented.
- An array allocated via a placement new can be deallocated via delete.
- Covariant return types on overriding virtual functions are supported.
- `enum` types are considered to be non-integral types.
- Partial specialization of class templates is implemented.
- Partial ordering of function templates is implemented.
- Function declarations that match a function template are regarded as independent functions, not as “guiding declarations” that are instances of the template.

- It is possible to overload operators using functions that take enum types and no class types.
- Explicit specification of function template arguments is supported.
- Unnamed template parameters are supported.
- The new lookup rules for member references of the form `x.A::B` and `p->A::B` are supported.
- The notation `:: template` (and `->template`, etc.) is supported.
- In a reference of the form `f()->g()`, with `g` a static member function, `f()` is evaluated, and likewise for a similar reference to a static data member. The ARM specifies that the left operand is not evaluated in such cases.
- `enum` types can contain values larger than can be contained in an `int`.
- Default arguments of function templates and member functions of class templates are instantiated only when the default argument is used in a call.
- String literals and wide string literals have `const` type.
- Class name injection is implemented.
- Argument-dependent (Koenig) lookup of function names is implemented.
- Class and function names declared only in unqualified friend declarations are not visible except for functions found by argument-dependent lookup.
- A `void` expression can be specified on a return statement in a `void` function.
- Universal character set escapes (e.g., `\uabcd`) are implemented.
- On a call in which the expression to the left of the opening parenthesis has class type, overload resolution looks for conversion functions that can convert the class object to pointer-to-function types, and each such pointed-to "surrogate function" type is evaluated alongside any other candidate functions.

New Language Features Not Accepted

The following features of the C++ standard are not implemented yet:

- `reinterpret_cast` does not allow casting a pointer to member of one class to a pointer to member of another class if the classes are unrelated.
- Two-phase name binding in templates, as described in [temp.res] and [temp.dep] of the standard, is not implemented.
- Template template parameters are not implemented.
- The `export` keyword for templates is not implemented.
- A `typedef` of a function type cannot include member function cv-qualifiers.

- A partial specialization of a class member template cannot be added outside of the class definition.

Anachronisms Accepted

The following anachronisms are accepted when anachronisms are enabled:

- `overload` is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array may be specified in an array `delete` operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name may be omitted in a base class initializer if there is only one immediate base class.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as a nonnested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-const type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-const class type may be initialized from an rvalue of the class type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following declares the overloading of two functions named `f`:

```
int f(int);
int f(x) char x; { return x; }
```

- It will be noted that in C this code is legal but has a different meaning: a tentative declaration of `f` is followed by its definition.
- When `--nonconst_ref_anachronism` is enabled, a reference to a nonconst class can be bound to a class rvalue of the same type or a derived type thereof.

```
struct A {
  A(int);
  A operator=(A&);
  A operator+(const A&);
};
main () {
  A b(1);
  b = A(1) + A(2);   // Allowed as anachronism
}
```

Extensions Accepted in Normal C++ Mode

The following extensions are accepted in all modes (except when strict ANSI violations are diagnosed as errors):

- A `friend` declaration for a class may omit the `class` keyword:

  ```
  class B;
  class A {
    friend B;  // Should be "friend class B"
  };
  ```

- Constants of scalar type may be defined within classes (this is an old form; the modern form uses an initialized static data member):

  ```
  class A {
    const int size = 10;
    int a[size];
  };
  ```

- In the declaration of a class member, a qualified name may be used:

  ```
  struct A {
    int A::f();  // Should be int f();
  };
  ```

- The preprocessing symbol `c_plusplus` is defined in addition to the standard `__cplusplus`.

- An extension is supported to allow an anonymous union to be introduced into a containing class by a `typedef` name — it needn't be declared directly, as with a true anonymous union. For example:

  ```
  typedef union {
    int i, j;
  } U;    // U identifies a reusable anonymous union.
  class A {
    U;               // Okay -- references to A::i and
                     // A::j are allowed.
  };
  ```

  In addition, the extension also permits "anonymous classes" and "anonymous structs," as long as they have no C++ features (e.g., no static data members or member functions and no nonpublic members) and have no nested types other than other anonymous classes, structs, or unions. For instance,

  ```
  struct A {
    struct {
      int i, j;
  ```

```
  };                // Okay -- references to A::i and
                    // A::j are allowed.
};
```

- The NCEG proposed extension for C (see below) is itself extended to allow `restrict` as a type qualifier for reference and pointer-to-member types and for nonstatic member functions. The set of C++ extensions is described in J16/92-0057.

- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a "default" assignment operator — that is, such a declaration blocks the implicit generation of a copy assignment operator. (This is **`cfront`** behavior that is known to be relied upon in at least one widely used library.) Here's an example:

  ```
  struct A { };
  struct B : public A {
    B& operator=(A&);
  };
  ```

  By default, as well as in **`cfront`**-compatibility mode, there will be no implicit declaration of `B::operator=(const B&)`, whereas in strict-ANSI mode `B::operator=(A&)` is *not* a copy assignment operator and `B::operator=(const B&)` *is* implicitly declared.

- Implicit type conversion between a pointer to an `extern "C"` function and a pointer to an `extern "C++"` function is permitted. Here's an example:

  ```
  extern "C" void f();    // f's type has extern "C" 
  linkage
  void (*pf)()            // pf points to an extern 
  "C++" function
               = &f;       // error unless implicit 
  conversion is allowed
  ```

  It is disabled in strict-ANSI mode, unless you specify the option **`--implicit_extern_c_type_conversion`**.

- A "?" operator whose second and third operands are string literals or wide string literals can be implicitly converted to "`char *`" or "`wchar_t *`". (Recall that in C++ string literals are `const`. There is a deprecated implicit conversion that allows conversion of a string literal to "`char *`", dropping the `const`. That conversion, however, applies only to simple string literals. Allowing it for the result of a "?" operation is an extension.)

  ```
  char *p = x ? "abc" : "def";
  ```

- Except in strict-ANSI mode, default arguments may be specified for function parameters other than those of a top-level function declaration (e.g., they are accepted on `typedef` declarations and on pointer-to-function and pointer-to-member-function declarations).

Except where noted, all of the extensions described in the C dialect section are also allowed in C++ mode.

Extensions Accepted in cfront 2.1 Compatibility Mode

The following extensions are accepted in **`cfront`** 2.1 compatibility mode in addition to the extensions listed in the 2.1/3.0 section following (i.e., these are things that were corrected in the 3.0 release of **`cfront`**):

- The dependent statement of an `if`, `while`, `do-while`, or `for` is not considered to define a scope. The dependent statement may not be a declaration. Any objects constructed within the dependent statement are destroyed at exit from the dependent statement.
- Implicit conversion from integral types to enumeration types is allowed.
- A non-`const` member function may be called for a `const` object. A warning is issued.
- A `const void *` value may be implicitly converted to a `void *` value, e.g., when passed as an argument.
- When, in determining the level of argument match for overloading, a reference parameter is initialized from an argument that requires a non-class standard conversion, the conversion counts as a user-defined conversion. (This is an outright bug, which unfortunately happens to be exploited in the NIH class libraries.)
- When a builtin operator is considered alongside overloaded operators in overload resolution, the match of an operand of a builtin type against the builtin type required by the builtin operator is considered a standard conversion in all cases (e.g., even when the type is exactly right without conversion).
- A reference to a non-`const` type may be initialized from a value that is a `const`-qualified version of the same type, but only if the value is the result of selecting a member from a `const` class object or a pointer to such an object.
- A cast to an array type is allowed; it is treated like a cast to a pointer to the array element type. A warning is issued.
- When an array is selected from a class, the type qualifiers on the class object (if any) are not preserved in the selected array. (In the normal mode, any type qualifiers on the object are preserved in the element type of the resultant array.)
- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.
- An expression of type `void` may be supplied on the return statement in a function with a `void` return type. A warning is issued.
- A parameter of type "`const void *`" is allowed on `operator delete`; it is treated as equivalent to "`void *`".
- A period ("`.`") may be used for qualification where "`::`" should be used. Only "`::`" may be used as a global qualifier. Except for the global qualifier, the two kinds of qualifier operators may not be mixed in a given name (i.e., you may say `A::B::C` or `A.B.C` but not `A::B.C` or `A.B::C`). A

period may not be used in a vacuous destructor reference nor in a qualifier that follows a template reference such as `A<T>::B`.

- **`cfront`** 2.1 does not correctly look up names in friend functions that are inside class definitions. In this example function f should refer to the functions and variables (e.g., `f1` and `a1`) from the class declaration. Instead, the global definitions are used.

```
int a1;
int e1;
void f1();
class A {
  int a1;
  void f1();
  friend void f()
  {
    int i1 = a1;   // cfront uses global a1
    f1();          // cfront uses global f1
  }
};
```

  Only the innermost class scope is (incorrectly) skipped by **`cfront`** as illustrated in the following example.

```
int a1;
int b1;
struct A {
  static int a1;
  class B {
    static int b1;
    friend void f()
    {
      int i1 = a1;  // cfront uses A::a1
      int j1 = b1;  // cfront uses global b1
    }
  };
};
```

- `operator=` may be declared as a nonmember function. (This is flagged as an anachronism by **`cfront`** 2.1)

- A type qualifier is allowed (but ignored) on the declaration of a constructor or destructor. For example:

```
class A {
  A() const;         // No error in cfront 2.1 mode
};
```

Extensions Accepted in cfront 2.1 and 3.0 Compatibility Mode

The following extensions are accepted in both **`cfront`** 2.1 and **`cfront`** 3.0 compatibility mode (i.e., these are features or problems that exist in both **`cfront`** 2.1 and 3.0):

- Type qualifiers on the `this` parameter may to be dropped in contexts such as this example:

  ```
  struct A {
    void f() const;
  };
  void (A::*fp)() = &A::f;
  ```

  This is actually a safe operation. A pointer to a `const` function may be put into a pointer to non-`const`, because a call using the pointer is permitted to modify the object and the function pointed to will actually not modify the object. The opposite assignment would not be safe.

- Conversion operators specifying conversion to `void` are allowed.

- A nonstandard friend declaration may introduce a new type. A friend declaration that omits the elaborated type specifier is allowed in default mode, but in **`cfront`** mode the declaration is also allowed to introduce a new type name.

  ```
  struct A {
    friend B;
  };
  ```

- The third operand of the `?` operator is a conditional expression instead of an assignment expression as it is in the modern language.

- A reference to a pointer type may be initialized from a pointer value without use of a temporary even when the reference pointer type has additional type qualifiers above those present in the pointer value. For example,

  ```
  int *p;
  const int *&r = p;  // No temporary used
  ```

- A reference may be initialized with a null.

- Because **`cfront`** does not check the accessibility of types, access errors for types are issued as warnings instead of errors.

- When matching arguments of an overloaded function, a `const` variable with value zero is not considered to be a null pointer constant. In general, in overload resolution a null pointer constant must be spelled “`0`” to be considered a null pointer constant (e.g., `'\0'` is not considered a null pointer constant).

- Inside the definition of a class type, the qualifier in the declarator for a member declaration is dropped if that qualifier names the class being defined.

  ```
  struct S {
    void S::f(); // No warning with --microsoft_bugs
  };
  ```

- An alternate form of declaring pointer-to-member-function variables is supported, namely:

  ```
  struct A {
    void f(int);
  ```

```
  static void sf(int);
  typedef void A::T3(int);  // nonstd typedef decl
  typedef void T2(int);     // std typedef
};
typedef void A::T(int);  // nonstd typedef decl
T* pmf = &A::f;          // nonstd ptr-to-member decl
A::T2* pf = A::sf;      // std ptr to static mem decl
A::T3* pmf2 = &A::f;      // nonstd ptr-to-member
decl
```

where `T` is construed to name a routine type for a nonstatic member function of class `A` that takes an `int` argument and returns `void`; the use of such types is restricted to nonstandard pointer-to-member declarations. The declarations of `T` and `pmf` in combination are equivalent to a single standard pointer-to-member declaration:

```
void (A::* pmf)(int) = &A::f;
```

A nonstandard pointer-to-member declaration that appears outside of a class declaration, such as the declaration of `T`, is normally invalid and would cause an error to be issued. However, for declarations that appear within a class declaration, such as `A::T3`, this feature changes the meaning of a valid declaration. **`cfront`** version 2.1 accepts declarations, such as `T`, even when `A` is an incomplete type; so this case is also excepted.

- Protected member access checking is not done when the address of a protected member is taken. For example:

```
class B { protected: int i; };
class D : public B { void mf(); };
void D::mf() {
  int B::* pmi1 = &B::i;  // error, OK in cfront mode
  int D::* pmi2 = &D::i;  // OK
}
```

Note that protected member access checking for other operations (i.e., everything except taking a pointer-to-member address) is done in the normal manner.

- The destructor of a derived class may implicitly call the private destructor of a base class. In default mode this is an error but in **`cfront`** mode it is reduced to a warning. For example:

```
class A {
      ~A();
};
class B : public A {
      ~B();
};
B::~B(){}    // Error except in cfront mode
```

- When disambiguation requires deciding whether something is a parameter declaration or an argument expression, the pattern *type-name-or-keyword*`(`*identifier...*`)` is treated as an argument. For example:

```
class A { A(); };
double d;
```

```
A x(int(d));
A(x2);
```

By default `int(d)` is interpreted as a parameter declaration (with redundant parentheses), and so `x` is a function; but in **`cfront`**-compatibility mode `int(d)` is an argument and `x` is a variable.

The declaration `A(x2);` is also misinterpreted by **`cfront`**. It should be interpreted as the declaration of an object named `x2`, but in **`cfront`** mode is interpreted as a function style cast of `x2` to the type `A`.

Similarly, the declaration

```
int xyz(int());
```

declares a function named `xyz`, that takes a parameter of type "function taking no arguments and returning an `int`." In **`cfront`** mode this is interpreted as a declaration of an object that is initialized with the value `int()` (which evaluates to zero).

- A named bit-field may have a size of zero. The declaration is treated as though no name had been declared.
- Plain bit fields (i.e., bit fields declared with a type of `int`) are always unsigned.
- The name given in an elaborated type specifier is permitted to be a `typedef` name that is the synonym for a class name, e.g.,

```
typedef class A T;
class T *pa;               // No error in cfront mode
```

- No warning is issued on duplicate size and sign specifiers.

```
short short int i;       // No warning in cfront mode
```

- Virtual function table pointer update code is not generated in destructors for base classes of classes without virtual functions, even if the base class virtual functions might be overridden in a further-derived class. For example:

```
struct A {
  virtual void f() {}
  A() {}
  ~A() {}
};
struct B : public A {
  B() {}
  ~B() {f();}  // Should call A::f according to ARM
12.7
};
struct C : public B {
  void f() {}
} c;
```

In **`cfront`** compatibility mode, `B::~B` calls `C::f`.

- An extra comma is allowed after the last argument in an argument list, as for example in

```
f(1, 2, );
```

- A constant pointer-to-member-function may be cast to a pointer-to-function. A warning is issued.

```
struct A {int f();};
main () {
  int (*p)();
  p = (int (*)())A::f;  // Okay, with warning
}
```

- Arguments of class types that allow bitwise copy construction but also have destructors are passed by value (i.e., like C structures), and the destructor is not called on the “copy.” In normal mode, the class object is copied into a temporary, the address of the temporary is passed as the argument, and the destructor is called on the temporary after the call returns. Note that because the argument is passed differently (by value instead of by address), code like this compiled in **`cfront`** mode is not calling-sequence compatible with the same code compiled in normal mode. In practice, this is not much of a problem, since classes that allow bitwise copying usually do not have destructors.

- A union member may be declared to have the type of a class for which the user has defined an assignment operator (as long as the class has no constructor or destructor). A warning is issued.

- When an unnamed class appears in a `typedef` declaration, the `typedef` name may appear as the class name in an elaborated type specifier.

```
typedef struct { int i, j; } S;
struct S x;                // No error in cfront mode
```

- Two member functions may be declared with the same parameter types when one is static and the other is nonstatic with a function qualifier.

```
class A {
  void f(int) const;
  static void f(int);        // No error in cfront
mode
};
```

- The scope of a variable declared in the `for-init-statement` is the scope to which the `for` statement belongs.

```
int f(int i) {
  for (int j = 0; j < i; ++j) { /* ... */ }
  return j;                  // No error in cfront
mode
}
```

- Function types differing only in that one is declared `extern "C"` and the other `extern "C++"` can be treated as identical:

```
typedef void (*PF)();
extern "C" typedef void (*PCF)();
```

```
void f(PF);
void f(PCF);
```

- Functions declared `inline` have internal linkage.
- `enum` types are regarded as integral types.
- An uninitialized `const` object of non-POD class type is allowed even if its default constructor is implicitly declared:

```
struct A { virtual void f(); int i; };
const A a;
```

- A function parameter type is allowed to involve a pointer or reference to array of unknown bounds.
- If the user declares an `operator=` function in a class, but not one that can serve as the default `operator=`, and bitwise assignment could be done on the class, a default `operator=` is not generated; only the user-written `operator=` functions are considered for assignments (and therefore bitwise assignment is not done).
- A member function declaration whose return type is omitted (and thus implicitly `int`) and whose name is found to be that of a type is accepted if it takes no parameters:

```
typedef int I;
struct S {
  I(); // Accepted in cfront mode (declares "int
S::I()")
  I(int); // Not accepted
};
```

C Dialect Accepted

The front end accepts the ANSI C language as defined by X3.159–1989.

The special comments recognized by the UNIX `lint` program —

`/*ARGSUSED*/`

`/*VARARGS*/` (with or without a count of non-varying arguments)

`/*NOTREACHED*/`

— are also recognized by the front end.

C9X Extensions

Certain C language extensions that have been approved for inclusion in the forthcoming C9X language definition can be enabled selectively. Sometimes these extensions had been available in existing compilers under slightly different guises: options to enable some of these C9X-like extensions are also provided. Not all of the C9X extensions are available in C++ mode.

- When *not* compiling in strict ANSI C mode, end-of-line comments (using `//` as delimiter) are supported.
- The options `--variadic_macros`, `--no_variadic_macros`, `--extended_variadic_macros` and `--no_extended_variadic_macros` control whether macros taking a variable number of arguments are recognized. These are also available in C++ mode. The current default is to disallow them.

  Ordinary variadic macros (as included in the proposed C9X definition) are illustrated by the following example:

  ```
  #define OVM(x, ...) x(__VA_ARGS__)
  void f() { OVM(printf, "%s %d\n", "Three args for ",
  1); }
  /* Expands to: printf("%s %d\n", "Three args for ",
  1) */
  ```

  During expansion the special identifier `__VA_ARGS__` will be replaced by the trailing arguments of the macro invocation. If variadic macros are enabled, this special identifier can appear only in the replacement list of variadic macros.

  Extended variadic macros (as implemented by certain pre-C9X compiler) use a slightly different syntax and allow the name of the variadic parameter to be chosen (instead of `...`/`__VA_ARGS__`):

  ```
  #define EVM(x, args...) x(args)
  void f() { EVM(printf, "%s %d\n", "Three args for ",
  1); }
  /* Same expansion as previous example. */
  ```

  In addition, enabling extended variadic macros adds a special behavior to the token pasting operator ## when it is followed by an empty or omitted macro argument: the macro parameter or continuous sequence of non-whitespace characters (not part of a macro parameter) preceding the operator is erased. Hence,

  ```
  EVM("Hello World\n")
  ```

  expands to `printf("Hello World\n")` and the extraneous comma is erased.

  Enabling either kind of variadic macros also allows trailing macro arguments to be omitted:

  ```
  #define M(a, b)
  void M(f); /* Becomes: void f(); No error or warning.
  */
  ```

- If the **`--long_long`** option is specified,
  - the `long long` and `unsigned long long` types are accepted;

  - integer constants suffixed by `LL` are given the type `long long`, and those suffixed by `ULL` are given the type `unsigned long long` (any of the suffix letters may be written in lower case);
  - the specifier `%lld` is recognized in `printf` and `scanf` formatting strings; and
  - the `long long` types are accommodated in the usual arithmetic conversions.

- An extension is supported to allow `restrict` as a type qualifier for object pointer types and function parameter arrays that decay to pointers. Its presence is recorded in the IL so that back ends can perform optimizations that would otherwise be prevented because of possible aliasing. This extension follows the NCEG proposal for incorporating `restrict` into C (see X3J11.1 Technical Report 2).

- Designators may be accepted in initializers for aggregates. Designators are not allowed in C++ mode however. See also command line options `--designators`, `--no_designators`, `--extended_designators` and `--no_extended_designators`. Currently, they are disabled by default.

  When **`--designators`** is specified, designators of the forms `.x` and `[k]` are accepted. They can be concatenated to reach nested aggregate elements. For example:

  ```
  struct X { double a; int b[10] } x
         { .b = { 1, [5] = 2 }, .b[3] = 1, .a = 42.0 };
  ```

  In addition, when **`--extended_designators`** is used, designators of the form `x:` and `[m ... n]` are accepted and the assignment (=) token becomes optional after array element designators. Field designators of the form `x:` cannot immediately be followed by an assignment token (=) or another designator. Examples:

  ```
  struct X { double a; int b[10] } x
         { b: { 1, [5 ... 9] = 2 }, .b[7] 1, a: 42.0 };
  struct Y y = { b:[3] /* Error */ = 7,
                   a: = /* Error */ 42.0 };
  ```

  Designators permit multiple initializations of the same subobject: only the last value is retained, but side-effects of prior initializing expressions do occur.

  A future release will enable these options by default.

ANSI C Extensions

The following extensions are accepted:

- A translation unit (input file) can contain no declarations.
- Comment text can appear at the ends of preprocessing directives.
- `__ALIGNOF__` is similar to `sizeof`, but returns the alignment requirement value for a type, or `1` if there is no alignment requirement. It may be followed by a type or expression in parentheses:

```
__ALIGNOF__(type)
__ALIGNOF__(expression)
```

The expression in the second form is not evaluated.

- `__INTADDR__(`*expression*`)` scans the enclosed expression as a constant expression, and converts it to an integer constant (it is used in the `offsetof` macro).
- Bit fields may have base types that are `enum`s or integral types besides `int` and `unsigned int`. This matches A.6.5.8 in the ANSI Common Extensions appendix.
- The last member of a `struct` may have an incomplete array type. It may not be the only member of the struct (otherwise, the struct would have zero size). (Allowed also in C++, but only when the structure is C-like.)
- A file-scope array may have an incomplete `struct`, `union`, or `enum` type as its element type. The type must be completed before the array is subscripted (if it is), and by the end of the compilation if the array is not `extern`. In C++, an incomplete `class` is also allowed.
- Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.
- `enum` tags may be incomplete: one may define the tag name and resolve it (by specifying the brace-enclosed list) later.
- The values of enumeration constants may be given by expressions that evaluate to unsigned quantities that fit in the `unsigned int` range but not in the `int` range. A warning is issued for suspicious cases.

```
/* When ints are 32 bits: */
enum a {w = -2147483648};  /* No warning */
enum b {x = 0x80000000};   /* No warning */
enum c {y = 0x80000001};   /* No warning */
enum d {z = 2147483649};   /* Warning */
```

- An extra comma is allowed at the end of an `enum` list. A remark is issued except in `pcc` mode.
- The final semicolon preceding the closing `}` of a struct or union specifier may be omitted. A warning is issued except in `pcc` mode.
- A label definition may be immediately followed by a right brace. (Normally, a statement must follow a label definition.) A warning is issued.
- An empty declaration (a semicolon with nothing before it) is allowed. A remark is issued.
- An initializer expression that is a single value and is used to initialize an entire static array, struct, or union need not be enclosed in braces. ANSI C requires the braces.
- In an initializer, a pointer constant value may be cast to an integral type if the integral type is big enough to contain it.
- The address of a variable with `register` storage class may be taken. A warning is issued.

- In an integral constant expression, an integer constant may be cast to a pointer type and then back to an integral type.
- In duplicate size and sign specifiers (e.g., `short short` or `unsigned unsigned`) the redundancy is ignored. A warning is issued.
- `long float` is accepted as a synonym for `double`.
- Benign redeclarations of `typedef` names are allowed. That is, a typedef name may be redeclared in the same scope as the same type. A warning is issued.
- Dollar signs can be accepted in identifiers.
- Numbers are scanned according to the syntax for numbers rather than the `pp-number` syntax. Thus, `0x123e+1` is scanned as three tokens instead of one invalid token.
- Assignment and pointer difference are allowed between pointers to types that are interchangeable but not identical, for example, `unsigned char *` and `char *`. This includes pointers to same-sized integral types (e.g., typically, `int *` and `long *`). A warning is issued except in `pcc` mode. Assignment of a string constant to a pointer to any kind of character is allowed without a warning.
- Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level (e.g., `int **` to `const int **`). Comparisons and pointer difference of such pairs of pointer types are also allowed. A warning is issued.
- In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ANSI C, some operators allow such things, and others (generally, where it does not make sense) do not allow them.
- Pointers to different function types may be assigned or compared for equality (`==`) or inequality (`!=`) without an explicit type cast. A warning is issued. This extension is not allowed in C++ mode.
- A pointer to `void` may be implicitly converted to or from a pointer to a function type.
- The `#assert` preprocessing extensions of AT&T System V release 4 are allowed. These allow definition and testing of predicate names. Such names are in a name space distinct from all other names, including macro names. A predicate name is given a definition by a preprocessing directive of the form

  ```
  #assert name
  #assert name(token-sequence)
  ```

  which defines the predicate *name*. In the first form, the predicate is not given a value. In the second form, it is given the value *token-sequence*.

  Such a predicate can be tested in a `#if` expression, as follows

  ```
  #name(token-sequence)
  ```

which has the value 1 if a `#assert` of that *name* with that *token-sequence* has appeared, and 0 otherwise. A given predicate may be given more than one value at a given time.

A predicate may be deleted by a preprocessing directive of the form

`#unassert` *name*
`#unassert` *name*`(`*token-sequence*`)`

The first form removes all definitions of the indicated predicate name; the second form removes just the indicated definition, leaving any others there may be.

- An extension is supported to allow constructs similar to C++ anonymous unions, including the following:
  - not only anonymous unions but also anonymous structs are allowed — that is, their members are promoted to the scope of the containing struct and looked up like ordinary members;
  - they can be introduced into the containing struct by a `typedef` name — they needn't be declared directly, as with true anonymous unions; and
  - a tag may be declared (C mode only).

  Among the restrictions: the extension only applies to constructs within structs.
- External entities declared in other scopes are visible. A warning is issued.

```
void f1(void) { extern void f(); }
void f2() { f(); /* Using out of scope declaration */
}
```

- A non-lvalue array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.

In the following areas considered "undefined behavior" by the ANSI C standard, the front end does the following:

- Adjacent wide and non-wide string literals are not concatenated unless `wchar_t` and `char` are the same type. (In C++ mode, when `wchar_t` is a keyword, adjacent wide and non-wide string literals are never concatenated.)
- In character and string escapes, if the character following the `\` has no special meaning, the value of the escape is the character itself. Thus `"\s" == "s"`. A warning is issued.
- A `struct` that has no named fields but at least one unnamed field is accepted by default, but a diagnostic (a warning or error) is issued in strict ANSI C mode.

K&R/pcc Mode

When `pcc` mode is specified, the front end accepts the traditional C language defined by *The C Programming Language*, first edition, by Kernighan and Ritchie (K&R), Prentice-Hall, 1978. In addition, it provides almost complete compatibility with the Reiser `cpp` and Johnson `pcc` widely used as part of UNIX systems; since there is no documentation of the exact behavior of those programs, complete compatibility cannot be guaranteed.

In general, when compiling in `pcc` mode, the front end attempts to interpret a source program that is valid to `pcc` in the same way that `pcc` would. However, ANSI features that do not conflict with this behavior are not disabled.

In some cases where `pcc` allows a highly questionable construct, the front end will accept it but give a warning, where `pcc` would be silent (for example: `0x`, a degenerate hexadecimal number, is accepted as zero).

The known cases where the front end is not compatible with `pcc` are the following:

- Token pasting is not done outside of macro expansions (i.e., in the primary source line) when two tokens are separated only by a comment. That is, `a/**/b` is not considered to be `ab`. The `pcc` behavior in that case can be gotten by preprocessing to a text file and then compiling that file.

  The textual output from preprocessing is also equivalent but not identical: the blank lines and white space will not be exactly the same.
- `pcc` will consider the result of a `?:` operator to be an lvalue if the first operand is constant and the second and third operands are compatible lvalues. This front end will not.
- `pcc` mis-parses the third operand of a `?:` operator in a way that some programs exploit:

  ```
  i ? j : k += l
  ```

  is parsed by `pcc` as

  ```
  i ? j : (k += l)
  ```

  which is not correct, since the precedence of `+=` is lower than the precedence of `?:`. This front end will generate an error for that case.
- `lint` recognizes the keywords for its special comments anywhere in a comment, regardless of whether or not they are preceded by other text in the comment. The front end only recognizes the keywords when they are the first identifier following an optional initial series of blanks and/or horizontal tabs. `lint` also recognizes only a single digit of the `VARARGS` count; the front end will accumulate as many digits as appear.

The differences in `pcc` mode relative to the default ANSI mode are as follows:

- The keywords `signed`, `const`, and `volatile` are disabled, to avoid problems with items declared with those names in old-style code. Those keywords were ANSI C inventions. The other non-K&R keywords (`enum` and `void`) are judged to have existed already in code and are not disabled.

- Declarations of the form

  ```
  typedef some-type void;
  ```

  are ignored.

- Assignment is allowed between pointers and integers, and between incompatible pointer types, without an explicit cast. A warning is issued.

- A field selection of the form *p*`->`*field* is allowed even if *p* does not point to a `struct` or `union` that contains *field*. *p* must be a pointer or an integer. Likewise, *x*`.`*field* is allowed even if *x* is not a `struct` or `union` that contains *field*. *x* must be an lvalue. For both cases, all definitions of *field* as a field must have the same offset within their `struct` or `union`.

- Overflows detected while folding signed integer operations on constants will cause warnings rather than errors. Usually this should be set to match the desired target machine behavior on integer operations in C.

- Integral types with the same representation (size, signedness, and alignment) will be considered identical and may be used interchangeably. For example, this means that `int` and `long` will be interchangeable if they have the same size.

- A warning will be issued for a `&` applied to an array. The type of such an operation is “address of array element” rather than “address of array”.

- For the shift operators `<<` and `>>`, the usual arithmetic conversions are done, the right operand is converted to `int`, and the result type is the type of the left operand. In ANSI C, the integral promotions are done on the two operands, and the result type is the type of the left operand. The effect of this difference is that, in `pcc` mode, a `long` shift count will force the shift to be done as `long`.

- When preprocessing output is generated, the line-identifying directives will have the `pcc` form instead of the ANSI form.

- String literals will not be shared. Identical string literals will cause multiple copies of the string to be allocated.

- `sizeof` may be applied to bit fields; the size is that of the underlying type (e.g., `unsigned int`).

- lvalues cast to a type of the same size remain lvalues, except when they involve a floating-point conversion.

- When a function parameter list begins with a `typedef` identifier, the parameter list is considered prototyped only if the `typedef` identifier is followed by something other than a comma or right parenthesis:

  ```
  typedef int t;
  int f(t) {}       /* Old-style list */
  int g(t x) {}     /* Prototyped list, parameter x of
  type t */
  ```

  That is, function parameters are allowed to have the same names as `typedef`s. In the normal ANSI mode, of course, any parameter list that begins with a `typedef` identifier is considered prototyped, so the first example above would give an error.

- The names of functions and of external variables are always entered at the file scope.
- A function declared `static`, used, and never defined is treated as if its storage class were `extern`.
- A file-scope array that has an unspecified storage class and remains incomplete at the end of the compilation will be treated as if its storage class is `extern` (in ANSI mode, the number of elements is changed to 1, and the storage class remains unspecified).
- The empty declaration

  ```
  struct x;
  ```

  will not hide an outer-scope declaration of the same tag.
- In a declaration of a member of a `struct` or `union`, no diagnostic is issued for omitting the declarator list; nevertheless, such a declaration has no effect on the layout. For example,

  ```
  struct s {char a; int; char b[2];} v;
                            /* sizeof(v) is 3 */
  ```

- `enum`s are always given type `int`. In ANSI mode, smaller integral types will be used if possible.
- No warning is generated for a storage specifier appearing in other than the first position in a list of specifiers (as in `int static`).
- `short`, `long`, and `unsigned` are treated as "adjectives" in type specifiers, and they may be used to modify a `typedef` type.
- A "plain" `char` is considered to be the same as `unsigned char` unless modified by command-line options. In ANSI C, "plain" `char` is a third type distinct from both `signed char` and `unsigned char`.
- Free-standing tag declarations are allowed in the parameter declaration list for a function with old-style parameters.
- `float` function parameters are promoted to `double` function parameters.
- `float` functions are promoted to `double` functions.
- Declaration specifiers are allowed to be completely omitted in declarations (ANSI C allows this only for function declarations). Thus

  ```
  i;
  ```

  declares `i` as an `int` variable. A warning is issued.
- All `float` operations are done as `double`.
- `__STDC__` is left undefined.
- Extra spaces to prevent pasting of adjacent confusable tokens are not generated in textual preprocessing output.

- The first directory searched for include files is the directory containing the file containing the `#include` instead of the directory containing the primary source file.
- Trigraphs are not recognized.
- Comments are deleted entirely (instead of being replaced by one space) in preprocessing output.
- `0x` is accepted as a hexadecimal `0`, with a warning.
- `1E+` is accepted as a floating-point constant with an exponent of `0`, with a warning.
- The compound assignment operators may be written as two tokens (e.g., `+=` may be written `+ =`).
- The digits `8` and `9` are allowed in octal constants.
- A warning rather than an error is issued for integer constants that are larger than can be accommodated in an `unsigned long`. The value is truncated to an acceptable number of low-order bits.
- The types of large integer constants are determined according to the K&R rules (they won't be `unsigned` in some cases where ANSI C would define them that way). Integer constants with apparent values larger than `LONG_MAX` are typed as `long` and are also marked as "non-arithmetic", which suppresses some warnings when using them.
- The escape `\a` (alert) is not recognized in character and string constants.
- Macro expansion is done differently. Arguments to macros are not macro-expanded before being inserted into the expansion of the macro. Any macro invocations in the argument text are expanded when the macro expansion is rescanned. With this method, macro recursion is possible and is checked for.
- Token pasting inside macro expansions is done differently. End-of-token markers are not maintained, so tokens that abut after macro substitution may be parsed as a single token.
- Macro parameter names inside character and string constants are recognized and substituted for.
- Macro invocations having too many arguments are flagged with a warning rather than an error. The extra arguments are ignored.
- Macro invocations having too few arguments are flagged with a warning rather than an error. A null string is used as the value of the missing parameters.
- Extra `#else`s (after the first has appeared in an `#if` block) are ignored, with a warning.
- Expressions in a `switch` statement are cast to `int`; this differs from the ANSI C definition in that a `long` expression is (possibly) truncated.
- The promotion rules for integers are different: `unsigned char` and `unsigned short` are promoted to `unsigned int`.

- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.

Extensions Accepted in SVR4 Compatibility Mode

The following extensions are accepted in SVR4 C compatibility mode:

- Macro invocations having too many arguments are flagged with a warning rather than an error. The extra arguments are ignored.
- Macro invocations having too few arguments are flagged with a warning rather than an error. A null string is used as the value of the missing parameters.
- The sequence `/**/` in a macro definition is treated as equivalent to the token-pasting operator ##.
- lvalues cast to a type of the same size remain lvalues, except when they involve a floating-point conversion.
- Assignment is allowed between pointers and integers, and between incompatible pointer types, without an explicit cast. A warning is issued.
- A field selection of the form *p*`->`*field* is allowed even if *p* does not point to a `struct` or `union` that contains *field*. *p* must be a pointer. Likewise, *x*`.`*field* is allowed even if *x* is not a `struct` or `union` that contains *field*. *x* must be an lvalue. For both cases, all definitions of *field* as a field must have the same offset within their `struct` or `union`.
- In an integral constant expression, an integer constant may be cast to a pointer type and then back to an integral type.
- Incompatible external object declarations are allowed if the object types share the same underlying representation.
- Certain incompatible function declarations are allowed. A warning is issued.

```
typedef unsigned int size_t;
extern size_t strlen(const char *);
extern int strlen();  /* Warning */
```

6
Special Features of C++

6
Special Features of C++

C++ provides powerful programming constructs. This chapter discusses the Concurrent C++ compiler's support of the following features:

- Namespace Support
- Template Instantiation
  - Automatic Instantiation
  - Instantiation Modes
  - Instantiation #pragma Directives
  - Implicit Inclusion
  - Automatic Instantiation Issues
- Predefined Macros
- Pragmas
- Precompiled Headers
- Intrinsic Functions
- AltiVec Technology Programming Interface
- Environment Variables
- Diagnostic Messages
- Termination Messages
- Response to Signals
- Exit Status
- Finding Include Files

Namespace Support

Namespaces are enabled by default except in the **`cfront`** modes. The command-line options **`--namespaces`** and **`--no_namespaces`** can be used to enable or disable the features.

Name lookup during template instantiations now does something that approximates the two-phase lookup rule of the standard. When a name is looked up as part of a template instantiation but is not found in the local context of the instantiation, it is looked up in a synthesized instantiation context. The front end follows the new instantiation lookup rules for namespaces as closely as possible in the absence of a complete implementation of the new template name binding rules. Here's an example:

```
namespace N {
  int g(int);
  int x = 0;
  template <class T> struct A {
    T f(T t) { return g(t); }
    T f() { return x; }
  };
}
namespace M {
  int x = 99;
  double g(double);
  N::A<int> ai;
  int i = ai.f(0);        // N::A<int>::f(int) calls N::g(int)
  int i2 = ai.f();        // N::A<int>::f() returns 0 (= N::x)
  N::A<double> ad;
  double d = ad.f(0);     // N::A<double>::f(double) calls
M::g(double)
  double d2 = ad.f();     // N::A<double>::f() also returns 0 (=
N::x)
}
```

The lookup of names in template instantiations does not conform to the rules in the standard in the following respects:

- Although only names from the template definition context are considered for names that are not functions, the lookup is not limited to those names visible at the point at which the template was defined.
- Functions from the context in which the template was referenced are considered for all function calls in the template. Functions from the referencing context should only be visible for "dependent" function calls.

For details of the algorithm implemented, see the Symbol Table chapter (in particular the section entitled "Instantiation Context Lookup").

The lookup rules for overloaded operators are implemented as specified by the standard, which means that the operator functions in the global scope overload with the operator functions declared extern inside a function, instead of being hidden by them. The old operator function lookup rules are used when namespaces are turned off. This means a program can have different behavior, depending on whether it is compiled with namespace support enabled or disabled:

```
struct A { };
A operator+(A, double);
void f() {
  A a1;
  A operator+(A, int);
  a1 + 1.0;          // calls operator+(A, double) with
}                    // namespaces enabled but otherwise
                     // calls operator+(A, int);
```

Template Instantiation

The C++ language includes the concept of *templates*. A template is a description of a class or function that is a model for a family of related classes or functions.[1] For example, one can write a template for a `Stack` class, and then use a stack of integers, a stack of floats, and a stack of some user-defined type. In the source, these might be written `Stack<int>`, `Stack<float>`, and `Stack<X>`. From a single source description of the template for a stack, the compiler can create *instantiations* of the template for each of the types required.

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, for several reasons:

- One would like to end up with only one copy of each instantiated entity across all the object files that make up a program. (This of course applies to entities with external linkage.)
- The language allows one to write a *specialization* of a template entity, i.e., a specific version to be used in place of a version generated from the template for a specific data type. (One could, for example, write a version of `Stack<int>`, or of just `Stack<int>::push`, that replaces the template-generated version; often, such a specialization provides a more efficient representation for a particular data type.) Since the compiler cannot know, when compiling a reference to a template entity, if a specialization for that entity will be provided in another compilation, it cannot do the instantiation automatically in any source file that references it. (The modern C++ language requires that a specialization be declared in every compilation in which it is used, but for compatibility with existing code and older compilers the Concurrent compiler does not require that in some modes. See the command-line option **`--no_distinct_template_signatures`**.)
- The language also dictates that template functions that are not referenced should not be compiled, that, in fact, such functions might contain semantic errors that would prevent them from being compiled. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.

(It should be noted that certain template entities are always instantiated when used, e.g., inline functions.)

From these requirements, one can see that if the compiler is responsible for doing all the instantiations automatically, it can only do so on a program-wide basis. That is, the compiler cannot make decisions about instantiation of template entities until it has seen all the source files that make up a complete program.

The C++ front end provides an instantiation mechanism that does automatic instantiation at link time. For cases where the programmer wants more explicit control over instantia-

1. Since templates are descriptions of entities (typically, classes) that are parameterizable according to the types they operate upon, they are sometimes called *parameterized types*.

tion, the front end also provides instantiation modes and instantiation pragmas, which can be used to exert fine-grained control over the instantiation process. The Program Development Environment (PDE) tools handle template instantiation automatically, but provide tools for manipulating how the instantiation happens.

Automatic Instantiation

The goal of an automatic instantiation mode is to provide painless instantiation. The programmer should be able to compile source files to object code, then link them and run the resulting program, and never have to worry about how the necessary instantiations get done.

In practice, this is hard for a compiler to do, and different compilers use different automatic instantiation schemes with different strengths and weaknesses:

- AT&T/USL/Novell/SCO's **`cfront`** product saves information about each file it compiles in a special directory called `ptrepository`. It instantiates nothing during normal compilations. At link time, it looks for entities that are referenced but not defined, and whose mangled names indicate that they are template entities. For each such entity, it consults the `ptrepository` information to find the file containing the source for the entity, and it does a compilation of 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.

  The programmer using **`cfront`** must follow a particular coding convention: all templates must be declared in "**`.h`**" files, and for each such file there must be a corresponding "**`.C`**" file containing the associated definitions. The compiler is never told about the "**`.C`**" files explicitly; one does not, for example, compile them in the normal way. The link step looks for them when and if it needs them, and does so by taking the "**`.h`**" file name and replacing its suffix.[2]

  This scheme has the disadvantage that it does a separate compilation for each instantiated function (or, at best, one compilation for all the member functions of one class). Even though the function itself is often quite small, it must be compiled along with the declarations for the types on which the instantiation is based, and those declarations can easily run into many thousands of lines. For large systems, these compilations can take a very long time. The link step tries to be smart about recompiling instantiations only when necessary, but because it keeps no fine-grained dependency information, it is often forced to "recompile the world" for a minor change in a "**`.h`**" file. In addition, **`cfront`** has no way of ensuring that preprocessing symbols are set correctly when it does these instantiation compilations, if preprocessing symbols are set other than on the command line.

- Borland's C++ compiler instantiates everything referenced in a compilation, then uses a special linker to remove duplicate definitions of instantiated functions.

2. The actual implementation allows for several different suffixes and provides a command-line option to change the suffixes sought.

The programmer using Borland's compiler must make sure that every compilation sees all the source code it needs to instantiate all the template entities referenced in that compilation. That is, one cannot refer to a template entity in a source file if a definition for that entity is not included by that source file. In practice, this means that either all the definition code is put directly in the "**`.h`**" files, or that each "**`.h`**" file includes an associated "**`.C`**" (actually, "**`.CPP`**") file.

This scheme is straightforward, and works well for small programs. For large systems, however, it tends to produce very large object files, because each object file must contain object code (and symbolic debugging information) for each template entity it references.

Concurrent's approach is a little different. It requires that, for each instantiation, there is some (normal, top-level, explicitly-compiled) source file that contains the definition of the template entity, a reference that causes the instantiation, and the declarations of any types required for the instantiation.[3] This requirement can be met in various ways:

- The Borland convention: each "**`.h`**" file that declares a template entity also contains either the definition of the entity or includes another file containing the definition.
- Implicit inclusion: when the compiler sees a template declaration in a "**`.h`**" file and discovers a need to instantiate that entity, it is given permission to go off looking for an associated definition file having the same base name and a different suffix, and it implicitly includes that file at the end of the compilation. This method allows most programs written using the **`cfront`** convention to be compiled with Concurrent's approach. See the section on implicit inclusion.
- The ad hoc approach: the programmer makes sure that the files that define template entities also have the definitions of all the available types, and adds code or pragmas in those files to request instantiation of the entities there.

The Concurrent automatic instantiation method works as follows:[4]

1. The first time the source files of a program are compiled, no template entities are instantiated. However, template information files (with, by default, a "**`.ti`**" suffix) are generated and contain information about things that *could* have been instantiated in each compilation. When compilation is done through the PDE tools or NightBench, the template information is placed in the environment's database.
2. When the object files are linked together, 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. Isn't this always the case? No. Suppose that file A contains a definition of class `X` and a reference to `Stack<X>::push`, and that file B contains the definition for the member function `push`. There would be no file containing both the definition of `push` and the definition of `X`.

4. It should be noted that automatic instantiation, more than most aspects of the C++ language, requires environmental support outside of the compiler. This is likely to be operating-system and object-format dependent.

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 is recorded in an associated instantiation request file (with, by default, a "**`.ii`**" suffix). When compilation is done through the PDE tools or NightBench, the instantiation requests are recorded in the environment's database.

4. The prelinker then executes the compiler again to recompile each file for which the instantiation request file was changed. The original compilation command-line options (saved in the template information file) are used for the recompilation.

5. When the compiler compiles a file, it reads the instantiation request file or consults the PDE's database for that file and obeys the 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). The compiler also receives a definition list file, which lists all the instantiations for which definitions already exist in the set of object files. If during the compilation the compiler has the opportunity to instantiate a referenced entity that is not on that list, it goes ahead and does the instantiation. It passes back to the prelinker (in the definition list file) a list of the instantiations that it has "adopted" in this way, so the prelinker can assign them to the file. This adoption process allows rapid instantiation and assignment of instantiations referenced from new instantiations, and reduces the need to recompile a given file more than once during the prelinking process.

6. The prelinker repeats steps 3–5 until there are no more instantiations to be adjusted.

7. The object files are linked together.

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

If the programmer provides 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 instantiation request file.

The instantiation request files should not, in general, require any manual intervention. One exception: if a definition is changed in such a way that some instantiation no longer compiles (it gets errors), and at the same time a specialization is added in another file, and the first file is being recompiled before the specialization file and is getting errors, the instantiation request file for the file getting the errors must be deleted manually to allow the prelinker to regenerate it. Should such a situation arise in the PDE, use the **`c.resolve`** with the **`-u`** option to manually remove the association between a template instantiation and a compilation unit.

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

```
C++ prelinker: A<int>::f() assigned to file test.o
C++ prelinker: executing: /edg/bin/eccp -c test.c
```

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. See the following sections.

Instantiations are normally generated as part of the object file of the translation unit in which the instantiations are performed. But when “one instantiation per object” mode is specified, each instantiation is placed in its own object file. One-instantiation-per-object mode is useful when generating libraries that need to include copies of the instances referenced from the library. If each instance is not placed in its own object file, it may be impossible to link the library with another library containing some of the same instances. Without this feature it is necessary to create each individual instantiation object file using the manual instantiation mechanism.

The automatic instantiation mode can be turned on or off using the **`--auto_instantiation`** and **`--no_auto_instantiation`** command-line options. If automatic instantiation is turned off, the template information file is not generated.

Instantiation Modes

Normally, when a file is compiled, no template entities are instantiated (except those assigned to the file by automatic instantiation). The overall instantiation mode can, however, be changed by a command line option:

`-tnone` Do not automatically create instantiations of any template entities. This is the default. It is also the usually appropriate mode when automatic instantiation is done.

`-tused` Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions.

`-tall` Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Nonmember template functions will be instantiated even if the only reference was a declaration.

`-tlocal` Similar to **`-tused`** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables.) However, one may end up with many copies of the instantiated functions, so this is not suitable for production use. **`-tlocal`** can not be used in conjunction with automatic template instantiation. If automatic instantiation is enabled by default, it will be disabled by the **`-tlocal`** option. If automatic instantiation is not enabled by default, use of **`-tlocal`** and **`-T`** is an error.

In the case where the **`eccp`** script is given a single file to compile and link, e.g.,

```
eccp t.c
```

the compiler knows that all instantiations will have to be done in the single source file. Therefore, it uses the **`-tused`** mode and suppresses automatic instantiation.

Instantiation `#pragma` Directives

Instantiation pragmas can be used to control the instantiation of specific template entities or sets of template entities. There are three instantiation pragmas:

- The `instantiate` pragma causes a specified entity to be instantiated.
- The `do_not_instantiate` pragma suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition will be supplied.
- The `can_instantiate` pragma indicates that a specified entity can be instantiated in the current compilation, but need not be; it is used in conjunction with automatic instantiation, to indicate potential sites for instantiation if the template entity turns out to be required.[5]

The argument to the instantiation pragma may be:

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

A pragma in which the argument is a template class name (e.g., `A<int>` or `class A<int>`) is equivalent to repeating the pragma for each member function and static data member declared in the class. When instantiating an entire class a given member function or static data member may be excluded using the `do_not_instantiate` pragma. For example,

5. At the moment, the `can_instantiate` pragma ends up forcing the instantiation of the template instance even if it isn't referenced somewhere else in the program; that's a weakness of the initial implementation which we expect to address.

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

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the `instantiate` pragma and no template definition is available or a specific definition is provided, an error is issued.

```
template <class T> void f1(T);  // No body provided
template <class T> void g1(T);  // No body provided
void f1(int) {}  // Specific definition
void main()
{
  int     i;
  double  d;
  f1(i);
  f1(d);
  g1(i);
  g1(d);
}
#pragma instantiate void f1(int) // error - specific definition
#pragma instantiate void g1(int)  // error - no body provided
```

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

A member function name (e.g., `A<int>::f`) can only be used as a pragma argument if it refers to a single user defined member function (i.e., 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. Overloaded member functions can be instantiated by providing the complete member function declaration, as in

```
#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.

Implicit Inclusion

When implicit inclusion is enabled, the front end is given permission to assume that if it needs a definition to instantiate a template entity declared in a "**`.h`**" file it can implicitly include the corresponding "**`.C`**" file to get the source code for the definition. For example, if a template entity `ABC::f` is declared in file **`xyz.h`**, and an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears in the source code processed by the compilation, the compiler will look to see if a file **`xyz.C`** exists, and if so it will process it as if it were included at the end of the main source file.

To find the template definition file for a given template entity the front end needs to know the full path name of the file in which the template was declared and whether the file was included using the system include syntax (e.g., `#include <file.h>`). This information is not available for preprocessed source containing `#line` directives. Consequently, the front end will not attempt implicit inclusion for source code containing `#line` directives.

The set of definition-file suffixes tried is "**.c**", "**.C**", "**.cpp**", "**.CPP**", "**.cxx**", "**.CXX**", and "**.cc**".

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.

The implicit inclusion mode can be on or off using the **--implicit_include** and **--no_implicit_include** command-line options.

Implicit inclusions are only performed during the normal compilation of a file, (i.e., not when doing only preprocessing). A common means of investigating certain kinds of problems is to produce a preprocessed source file that can be inspected. When using implicit inclusion it is sometimes desirable for the preprocessed source file to include any implicitly included files. This may be done using the **--no_preproc_only** command line option. This causes the preprocessed output to be generated as part of a normal compilation. When implicit inclusion is being used, the implicitly included files will appear as part of the preprocessed output in the precise location at which they were included in the compilation.

Automatic Instantiation Issues

Template instantiation is a complex issue. We have been improving support for it, but it remains intrinsically confusing. This section will attempt to explain some of the issues surrounding template instantiation and its automation. It also documents enhancements that have been made in the 5.1 and 5.2 releases to better automate template instantiation.

The compiler can't know which templates will need to be instantiated in a program until link time. Therefore, when first compiling source files, no templates are instantiated unless their instantiation is explicitly requested in the source (or via command line options). The template instantiations that can be provided by and that are needed by each compilation are recorded in a **.ti** file that is placed in the same directory as the generated object file if the file is compiled with the **--auto_instantiation** option. Before actually linking, a tool called **prelink** collects a list of all the template instantiations that are needed to link successfully, and assigns each to a compilation that can provide the instantiation if the link is performed with the **--auto_instantiation** option. These assignments are recorded in **.ii** files in the same directory as the **.ti** and object files. It then recompiles those compilations and finally the linker is invoked.

The Problem

There is one obvious, huge, problem with this scheme. That is, if the build procedure moves the object file elsewhere, such as into an archive, then the prelinker has no way of finding the **.ti** and **.ii** files. The 5.1 release improved this situation **if** you use the program development environment tools (PDE tools). This is because the PDE's database knows how the archives were built, so it can assign a template instantiation to an compilation unit in an archive and update the archive before linking the program that uses the archive.

Consider the following **Makefile**:

```
pgm: main.o fg.a
        ec++ --auto_instantiation -o pgm main.o fg.a

main.o: main.c
        ec++ -c --auto_instantiation main.c

fg.a: f.o g.o
        ar r fg.a f.o g.o

f.o: f.c
        ec++ -c --auto_instantiation f.c

g.o: g.c
        ec++ -c --auto_instantiation g.c
```

If **`f.o`** and **`g.o`** require templates to be instantiated and **`main.o`** cannot provide the instantiations, then this program cannot be built because **`prelink`** does not know where to find the **`f.ti`**, **`f.ii`**, **`g.ti`**, and **`g.ii`** files.

Solving the Problem with `--prelink_objects`

The **`--prelink_objects`** option of **`ec++`** can be used to resolve the template instantiations needed by a subset of the object files. To use this in the above example, one would change the **`fg.a`** target to read:

```
fg.a: f.o g.o
        ec++ --prelink_objects f.o g.o
        ar r fg.a f.o g.o
```

This will direct the compiler to determine what templates are needed by **`f.o`** and **`g.o`**, assign instantiations of them to one or the other of them, if possible, and recompile them with those instantiations. There is a drawback however.

Consider the following more complex **`Makefile`**:

```
pgm: main.o fg.a hi.a
        ec++ --auto_instantiation -o pgm main.o fg.a hi.a

main.o: main.c
        ec++ -c --auto_instantiation main.c

fg.a: f.o g.o
        ec++ --prelink_objects f.o g.o
        ar r fg.a f.o g.o

f.o: f.c
        ec++ -c --auto_instantiation f.c

g.o: g.c
        ec++ -c --auto_instantiation g.c

hi.a: h.o i.o
        ec++ --prelink_objects h.o i.o
        ar r hi.a h.o i.o

h.o: h.c
        ec++ -c --auto_instantiation h.c

i.o: i.c
        ec++ -c --auto_instantiation i.c
```

What happens if **`f.o`** and **`h.o`** both require the same template instantiation? When both archives are prelinked, they will both instantiate that same template, and a duplicate symbol error will occur at link time.

The solution for this is to arrange for the template instantiations to be placed in separate object files using the **`--one_instantiation_per_object`** option on both the compilation, prelinking and linking commands. This way, only one of the instantiations get loaded. The following **`Makefile`** demonstrates how this could be set up. This example also uses the **`--instantiation_dir`** option to specify where the instantiations are to be placed.

```
CFLAGS=--auto_instantiation --one_instantiation_per_object

pgm: main.o fg.a hi.a
        ec++ $(CFLAGS) -o pgm main.o fg.a hi.a

main.o: main.c
        ec++ -c $(CFLAGS) main.c

fg.a: f.o g.o
        ec++ --prelink_objects $(CFLAGS) f.o g.o
        ar r fg.a f.o g.o fg/*.o

f.o: f.c
        ec++ -c $(CFLAGS) --instantiation_dir=fg f.c

g.o: g.c
        ec++ -c $(CFLAGS) --instantiation_dir=fg g.c

hi.a: h.o i.o
        ec++ --prelink_objects $(CFLAGS) h.o i.o
        ar r hi.a h.o i.o hi/*.o

h.o: h.c
        ec++ -c $(CFLAGS) --instantiation_dir=hi h.c

i.o: i.c
        ec++ -c $(CFLAGS) --instantiation_dir=hi i.c
```

This will place the instantiation assigned to **`f.o`** in the directory **`fg`**, which then gets archived into the archive **`fg.a`**. Similarly, the instantiation assigned to **`h.o`** is placed in the directory **`hi`** and is archived into the archive **`hi.a`**. When we link, the linker will pick up the instantiation from one of the archives, and not from the other, so we link without multiply defined symbols.

There is, of course, a caveat even with this scheme. Consider the situation where **`main.o`** does not reference **`f.o`**, **`g.o`** also requires the template to be instantiated, and the template uses a file scoped static variable (bad programming practice to be sure, but perfectly legal). The template instantiation has been assigned to **`f.o`**, so it is going to reference **`f.o`**'s file scoped static variable. To do this, a "mangled" external name is created for it. Now when we link, should the instance of the template assigned to **`f.o`** be linked with **`g.o`**, then to resolve the reference to the file scoped static variable, **`f.o`** will also be linked in, even though **`f.o`** is not otherwise needed. This will generally be harmless, except that it inflates program size.

Solving the Problem with the PDE Tools

Using the PDE tools, one would set up this environment like this (the equivalent actions can be done through NightBench graphical user interface):

```
c.mkenv
c.options -set -default -- --auto_instantiation
c.intro main.c f.c g.c h.c i.c
c.partition -create archive -add "f g" fg.a
c.partition -create archive -add “h i” hi.a
c.partition -create exe -add main -parts “fg.a hi.a” pgm
```

The environment's database knows all about the **`fg.a`** and **`hi.a`** archives and how to build them. When **`pgm`** is built using the **`c.build`** command, **`f.c`**, **`g.c`**, **`h.c`**, **`i.c`**, and

`main.c` will all be compiled, the archives will be created, then during the prelinking stage, template instantiations will be assigned to **`f.c`** or one of the other compilation units, they will be recompiled, the archive updated, and finally **`pgm`** will link without incident.

There is a caveat with not using the **`--one_instantiation_per_object`** option (this applies to using a **`Makefile`** too). If you have three executables that each use a pair of three compilation units and all three compilation units require the same template to be instantiated, there is no way to link the three programs without using **`--one_instantiation_per_object`**. If to link one program, the instantiation is assigned to one compilation unit, then the program that links in the other two compilation unit must force the instantiation to be assigned to one of them. This will cause either the first or the third program to get a multiply defined symbol on the template. This problem can be avoided by issuing the following command:

```
c.options -add -default -- --one_instantiation_per_object
```

Solving the Problem with Makefiles and the PDE Tools

Now, it is realized that for portability reasons, customers may not be willing to abandon their **`Makefile`**s. The 5.2 release has two enhancements to deal with this. The first is the **`c.make`** tool. This tool generates a **`Makefile`** from a PDE environment, making it possible to take a program that builds under the PDE on a Concurrent machine, and compile it elsewhere.

The second enhancement is in the invokers for **`ec`**, **`ec++`**, and **`ar`** that allow an existing **`Makefile`** (or any other program building mechanism) to build a program in the context of a PDE environment. These new invokers are activated by setting the `PDE_ENVIRONMENT` environment variable, or, instead, by placing a file called **`.pde_environment`** containing a single line of text specifying the path to the PDE environment to use in the directory where the compiler will be run.

The following is a slightly modified version of the above **Makefile**:

```
CFLAGS=--auto_instantiation --one_instantiation_per_object

pgm: main.o fg.a hi.a
        ec++ $(CFLAGS) -o pgm main.o fg.a hi.a

main.o: main.c
        ec++ -c $(CFLAGS) main.c

fg.a: f.o g.o
        ar r fg.a f.o g.o

f.o: f.c
        ec++ -c $(CFLAGS) f.c

g.o: g.c
        ec++ -c $(CFLAGS) g.c

hi.a: h.o i.o
        ar r hi.a h.o i.o

h.o: h.c
        ec++ -c $(CFLAGS) h.c

i.o: i.c
        ec++ -c $(CFLAGS) i.c
```

The explicit handling of the template instantiation object files has been removed because the PDE's database will handle all that for us automatically. The user may do the following:

```
mkdir pgm_env
mkenv -env pgm_env
export PDE_ENVIRONMENT=‘pwd‘/pgm_env
make
```

Now, when the **Makefile** invokes **ec++** on **f.c**, the **ec++** invoker, rather than directly invoking the 5.2 version of the compiler, will instead invoke the following commands:

```
c.intro -env pgm_env -language C++ -o f.o f.c
c.options -env pgm_env  -set -- --auto_instantiation \
  --one_instantiation_per_object f
c.compile -env pgm_env f
```

Subsequent invocations of **make** will result in **c.options** being invoked only if the options have changed. Normally, only **c.compile** needs to be invoked. Similar actions occur for the compilation of **g.c** and **main.c**. When the **ar** invoker gets invoked on **fg.a** for the first time, it will do the following:

```
c.partition -env pgm_env -create archive -add "f g" \
              -o fg.a fg.a
ar r fg.a f.o g.o
```

Finally, when **pgm** is linked for the first time, the **ec++** invoker will do this:

```
c.partition -env pgm_env -create executable \
              -add "main" -parts "fg.a hi.a" -o pgm pgm
c.build -env pgm_env pgm
```

Since **`c.build`** knows all about how the archives were constructed, it can do the template instantiations, update the archives if needed, and link **`pgm`** without a problem. Note that we have not explicitly placed the template instantiation objects into the archives. The environment is handling management of them. If it was desired for the `fg.a` to be prelinked and have the object files placed in it, the target would be constructed like this:

```
fg.a: f.o g.o
        ar r fg.a f.o g.o
        c.partition -oset “--prelink_objects” fg.a
```

Miscellaneous Notes

Once the program has been made with the **`Makefile`** once, the user may either continue using the **`Makefile`**, switch to using **`c.build`**, or switch to using NightBench (the graphical interface to the PDE). Note however, that if the **`Makefile`** does other actions, such as generate source files, or create object using other compilers or the assembler, or invoke **`ld`** directly, these actions are not known by the environment, and thus won't be performed if **`c.build`** or NightBench is used, unless the environment is manually modified to use the **`-make`** option of **`c.intro`** and **`c.options`** to escape to a **`Makefile`** to perform arbitrary actions before building a source file.

Another advantage of using this scheme is that the user can set temporary options without modifying the **`Makefile`**. For example, if the user wants to turn on the debug option for just **`f.c`** temporarily, then he can do the following:

```
c.options -temp -set -- -g f
rm f.o
make
```

When the **`c.compile`** tool gets invoked, it will use this temporary setting. If **`c.build`** is used instead of **`make`** to build the program, then removing **`f.o`** isn't necessary since the PDE tools know that **`f.o`** is out of date when an option gets changed (the **`make`** command doesn’t know this). Also the PDE tools know about all header file dependencies. When done with all the temporary settings, things can be reset to normal by issuing the command:

```
c.options -temp -clear all
```

Another advantage of the PDE environment is that NightBench can be used to examine what specific template instantiations were done and who did them. It also provides a way to manually override individual automatic decisions.

`Makefile`s (and scripts, etc.) can do things to thwart the `PDE_ENVIRONMENT` environment variable mechanism:

- If a relative path is placed in the variable, and the current directory is changed, then the environment obviously won’t be found.
- The mechanism uses the object file, archive, and executable’s output path to uniquely identify them. Relative paths are canonicalized to be relative to the specified PDE environment. If the user switches between relative and absolute paths, the first one used is the way it will be entered in the database. However, the mechanism will try a relative path if it fails to find an entity named with an absolute path, and vice versa.

- If an entity is referenced by different two different paths (because symbolic or hard links, for example), the mechanism will become confused and may not be able to perform template instantiation properly.
- If entities files are moved, renamed, or linked to, the mechanism won't recognize the new name as the entity it has already built. This generally won't prevent linking, but it may prevent template instantiation automation from functioning properly. This is a problem equivalent to putting object files into an archive without using the Program Development Environment.
- Removing object files after placing them in an archive is generally not a problem since c.build is not used to update the archive, and when linking, c.build is passed the -link option directing it to not recursivly update dependant partitions. However, template instantiation may recompile some units in an archive. These object files will not be removed after linking, so the user may be surprised to find some object files laying around that he thought were removed.

Predefined Macros

The front end defines a number of preprocessing macros. Many of them are only defined under certain circumstances. This section describes the macros that are provided and the circumstances under which they are defined.

`__STDC__`

Defined in ANSI C mode and in C++ mode. In C++ mode the value may be redefined.`__cplusplus`

Defined in C++ mode.

`c_plusplus`

Defined in default C++ mode, but not in strict mode.

`__STDC_VERSION__`

Defined in ANSI C mode with the value `199409L`. The name of this macro, and its value, are specified in Normative Addendum 1 of the ISO C Standard.

`__SIGNED_CHARS__`

Defined when plain `char` is signed. This is used in the `<limits.h>` header file to get the proper definitions of `CHAR_MAX` and `CHAR_MIN`.

`_WCHAR_T`

Defined in C++ mode when `wchar_t` is a keyword. The name of this predefined macro is specified by a configuration flag. `_WCHAR_T` is the default.

`_BOOL`

Defined in C++ mode when `bool` is a keyword. The name of this predefined macro is specified by a configuration flag. `_BOOL` is the default.

`__ARRAY_OPERATORS`

Defined in C++ mode when array new and delete are enabled. The name of this predefined macro is specified by a configuration flag. `__ARRAY_OPERATORS` is the default.

`__EXCEPTIONS`

Defined in C++ mode when exception handling is enabled.

`__RTTI`

Defined in C++ mode when RTTI is enabled.

`__PLACEMENT_DELETE`

Defined in C++ mode when placement delete is enabled.

`__EDG_RUNTIME_USES_NAMESPACES`

Defined in C++ mode.

`__EDG_IMPLICIT_USING_STD`

Defined in C++ mode when the **`--using_std`** command line option is set indicating that the standard header files should implicitly do a using-directive on the `std` namespace.

`__EDG__`

Always defined.

`__EDG_VERSION__`

Defined to an integral value that represents the version number of the front end. For example. version 2.42 is represented as 242.

`__embedded_cplusplus`

Defined as 1 in Embedded C++ mode.

`_ELF`

Defined for compiling for an ELF object file

`_IBM`

Defined for compiling to an IBM™ PowerPC™ based architecture

`_PPC`

Defined when compiling with the **`-Qtarget=ppc`** option.

`_PPC604`

Defined when compiling with the **`-Qtarget=ppc604`** option.

`_PPC604E`

Defined when compiling with the **`-Qtarget=ppc604e`** option.

`_PPC750`

Defined when compiling with the **`-Qtarget=ppc750`** option.

`_FAST_MATH_INTRINSICS`

Defined when compiling wiht the **`-F`** option

`unix`

Traditionally defined for all UNIX systems. This is not defined when compiling with the **`--strict`** option.

`_unix`

Alternate spelling for `unix`. This is not defined when compiling with the **`--strict`** option.

`_PowerMAXOS`

Defined to indicate the target operating system is PowerMAXOS.

`__HC__`

Defined to indicate that this is a Concurrent C/C++ compiler.

`__STDC__`

Defined in ANSI C mode and in C++ mode. The default value is 1 when compiling with the **`--strict`** option; 0, otherwise.

`__cplusplus`

Defined when compiling C++ code.

`c_plusplus`

Defined when compiling C++ code, but not when the **`--strict`** option is used.

`_STDC_VERSION_`

Defined in ANSI C mode with the value `199409L`. The name of this macro and its value are specified in the Normative Addendum 1 of the ISO C Standard.

`__SIGNED_CHARS__`

Defined when the **`--signed_chars`** option is used.

Pragmas

`#pragma` directives are used within the source program to request certain kinds of special processing. The `#pragma` directive is part of the standard C and C++ languages, but the meaning of any pragma is implementation-defined. The front end recognizes several pragmas.

Edison Defined Pragmas

The following are described in detail in the template instantiation section of this chapter:

```
#pragma instantiate
#pragma do_not_instantiate
#pragma can_instantiate
```

and two others are described in the section on precompiled header processing:

```
#pragma hdrstop
#pragma no_pch
```

The front end also recognizes `#pragma once`, which, when placed at the beginning of a header file, indicates that the file is written in such a way that including it several times has the same effect as including it once. Thus, if the front end sees `#pragma once` at the start of a header file, it will skip over it if the file is `#include`d again.

A typical idiom is to place an `#ifndef` guard around the body of the file, with a `#define` of the guard variable after the `#ifndef`:

```
#pragma once                // #ifndef FILE_H
#define FILE_H
... body of the header file ...
#endif
```

The `#pragma once` is marked as optional in this example, because the front end recognizes the `#ifndef` idiom and does the optimization even in its absence. `#pragma once` is accepted for compatibility with other compilers and to allow the programmer to use other guard-code idioms.

The `#pragma pack` is used to specify the maximum alignment allowed for nonstatic data members of structs and classes, even if that alignment is less than the alignment dictated by the member's type. The basic syntax is:

```
#pragma pack(n)
#pragma pack()
```

where argument *n*, a power-of-2 value, is the new packing alignment that is to go into effect for subsequent declarations, until another `#pragma pack` is seen. The second form cancels the effect of a preceding `#pragma pack(`*n*`)` and either restores the default packing alignment specified by the **`--pack_alignment`** command-line option or, if the option was not used, disables the packing of structs and classes. In addition, an enhanced syntax is also supported in which keywords `push` and `pop` can be used to manage a stack of packing alignment values — for instance:

```
#pragma pack (push, xxx)
#include "xxx.h"
#pragma pack (pop, xxx)
```

which has the effect saving the current packing alignment value, processing the include file (which may leave the packing alignment with an unknown setting), and restoring the original value.

In C++ a `#pragma pack` directive has “local” effect when it appears inside a function template, an instantiation of a class template, or the body of a member function or friend function that is defined within a class definition. When such a context is entered, the current pack alignment state is saved, so that it can be restored when the context is exited. For example,.

```
template <class T> class A {
  :
#pragma pack(1)
  :
};
#pragma pack(4)
main() {
  A<int> a;                      // #pragma pack(1) is local to A<int>
  struct S { char c; int i; };   // sizeof(S) == 8, not 5
}
```

Moreover, within an instantiation of a class or function template, the default pack alignment is that which is in effect at the point of the template’s definition, not at the point of its instantiation. Here is an example:

```
#pragma pack(1)
template <class T> int f(T) {
  struct S { char c; int i; };   // default pack alignment is 1 for
each
  return sizeof(S);          //   instance of f<T>
}
#pragma pack(4)
main() {
  int i = f(0);              // i = 5, not 8
}
```

These rules apply to inline-defined member functions, too, because their bodies are also scanned “out of order” relative to the textual order of the source program (i.e., not till all the class members have been declared).

```
#pragma pack(4)
struct A {
  int f() {
#pragma pack(1)                   // #pragma pack(1) is local to A::f
    struct S { char c; int i; };  // sizeof(S) is 5
  }
  int g() {
    struct S { char c; int i; };  // sizeof(S) is 8
  }
#pragma pack(1)
  int h() {
    struct S { char c; int i; };  // sizeof(S) is 5
  }
};
```

`#pragma ident` is recognized, as is `#ident`:

`#pragma ident "`*string*`"`
`#ident "`*string*`"`

Both are implemented by recording the string in a pragma entry and passing it to the back end.

`#pragma weak` is recognized. Its form is

`#pragma weak` *name1* `[ =` *name2* `]`

where *name1* is the name to be given "weak binding" and is a synonym for *name2* if the latter is specified. The entire argument string is recorded in the pragma entry and passed to the back end.

Concurrent Defined Pragmas

The directive `#pragma` communicates implementation defined directives to the compiler. Syntax.

`#pragma` *directive_string*

directive_string `::=`
    directive_w_poss_args
directive_w_poss_args `::=`
    directive_name `[`*argument* `[,`*argument* `... ]]`

The Concurrent implementation defined directives for use with `#pragma` appear below

Table 6-1. Implementation Defined Directives Used with #pragma

`align`	`ident`	`optimize_for_space`
`cautions`	`min_align`	`optimize_for_time`
`error`	`once`	`warnings`

Table 6-1. Implementation Defined Directives Used with #pragma (Cont.)

`errcount`	`opt_class`	`weak`
	`opt_level`	
`do_not_instantiate`	`can_instantiate`	`instantiate`

Source Listing Controls

The following message classes are supported by the C++ compiler.

`inform`	Advisory, issues such as generated code quality
`caution`	Advisory, like running **`lint(1)`**
`warning`	Probably error in program, compilation not aborted
`fatal`	Error in program, compilation will continue, no object produced
`abort`	Error in program, cannot continue compilation

The following directives control the format of the source listing, including error messages, produced by the C compiler.

`#pragma cautions {on | off}`

Enables or suppresses the printing of caution messages. The default is `off`. The same effect can be obtained by invoking **`cc++`** with the **`-n`** option.

`#pragma error` *errnum* `[`*errnum...*`]`

Controls the printing of certain error messages. *errnum* is the number displayed when the **`-display_error_number`** option is on, or the number 0, and must be preceded by a plus or minus sign (`+` or `-`). A minus sign suppresses printing of the message, while a plus sign enables printing. For example, the directive

```
#pragma error +25-36
```

enables error message 25 (“Uninitialized item”) but disables number 36 (“Undefined function”). An *errnum* of 0 refers to all selectable messages. For example,

```
#pragma error -0
```

suppresses the printing of all selectable error messages. The default for `error` is +0 (all messages are printed). The `error` directive does not affect fatal error messages; they are always printed.

`#pragma errcount {on | off}`

Specifies whether error messages disabled by the `error` directive should be included in the error totals at the end of the compilation. The default is `on` (all errors are included in the count).

`#pragma warnings {on | off}`

Enables or suppresses the printing of warning messages. Most warnings indicate that an error exists that prevents proper execution of the program. The default is `on`.

Listing control directives should occur immediately prior to the definition of the first function they affect.

Optimization Directives

These `#pragma` directives should be placed immediately before a function definition in the source code.

The directives that permit the programmer to control the amount and type of optimization the compiler uses are:

```
opt_level
opt_class
optimize_for_space
optimize_for_time
```

The `opt_level` directive controls the level of optimization. The syntax is:

```
#pragma opt_level { NONE | MINIMAL | GLOBAL | MAXIMAL | ULTIMATE}
```

The default is `MINIMAL`.

The `opt_class` directive controls the class of optimization. The syntax is:

```
#pragma opt_class { UNSAFE | SAFE | STANDARD }
```

`UNSAFE` is the default. If the `opt_class` is `UNSAFE`, then the compiler makes assumptions as to how the program was written in order to produce more efficient code. Currently, the only optimization affected by the `UNSAFE` class is algebraic simplification of expressions. The `opt_class` may be set to SAFE to disable unsafe optimizations. The `opt_class` may be set to `STANDARD` to provide unsafe optimizations allowed by the C++ language.

If the `optimize_for_space` directive is used, a smaller object file is created. A faster object file is created if the `optimize_for_time` directive is used. These directives are mutually exclusive; `optimize_for_time` is the default.

All optimization directives must occur outside function definitions. They remain in effect until explicitly altered. Optimization directives should occur immediately prior to the definition of the first function they should affect. See the "Program Optimization" chapter of the *Compilation Systems Volume 2* (*Concepts*) manual for more information on optimizations.

Optimization directives should occur immediately prior to the definition of the first function they affect.

Data Alignment Control Directives

The compiler supports pragmas that override the compiler's default choice of data alignment. This gives you some control over the size of structures and unions and control over the location of a structure's members. These pragmas do not cause the compiler to generate extra code for accessing data that is incorrectly aligned for the target machine; therefore, it is possible for you to specify alignments that will cause run-time errors.

Data Alignment Rules

Data alignment rules have been chosen to meet the hardware requirements of the target machine, to provide for fast access to data items, and to be as consistent as possible for all target machines supported by the Concurrent C++ compiler. Table 6-2 lists the current default alignments used by the compiler.

Table 6-2. Alignments by Data Type

Type	Alignment (bytes)
`[unsigned] char`	1
`signed char`	1
`bool`	1
`unsigned short [int]`	2
`[signed] short [int]`	2
`unsigned [int]`	4
`[signed] int`	4
`unsigned long [int]`	4
`[signed] long [int]`	4
`wchar_t`	4
`unsigned long long [int]`	8
`[signed] long long [int]`	8
`float`	4
`double`	8
`long double`	8
pointer	4

pointer indicates all pointer types. Array types have the alignment of their element type. The alignment of classes, structures, and unions is the maximum of the alignments of their members. In addition, the sizes of classes, structures, and unions are rounded up to the nearest multiple of the alignment size. See Appendix B (“Architecture Dependencies”) for more information on alignments.

#pragma align

The `align pragma` can be used to change the default minimum alignment of a given data type. It takes effect when it first appears in a file and remains in effect until changed by another `#pragma align`. The syntax for the `align` directive is:

```
#pragma align type n
```

where *n* is an integer constant that represents the minimum byte-alignment to use for the type, *type*. The type should be one of the following:

```
{char | short | int | long | pointer | float |
 double | struct | union}
```

The type `pointer` represents all pointer types.

When `struct` or `union` is used, the pragma specifies a minimum alignment to use for all structures or unions, respectively. The alignment of classes, structures, and unions is then the maximum of that value and of the alignments of the members.

This `pragma` also affects class, structure, and union bit-fields. When bit-fields are allocated for structures, they do not cross the alignment boundaries of the data type to which they were declared; therefore, changing the alignment of a bit-field's type may change its location in a structure.

The byte alignment value, *n*, must be between 0 and 31, inclusive. A value of 0 causes the compiler to reset the value to the default. If a byte value is chosen that can cause run-time exceptions (i.e., *n* is not a multiple of the minimum required alignment for the machine) a compiler warning message is produced, but the requested alignment is used. See "Bit-Field" on page B-1 for more information on bit-fields.

Due to the order in which the C++->C translator generates declarations and `pragma` directives, repeatedly modifying the alignment of a type may have unpredictable results.

#pragma min_align

Most C++ compilation units contain multiple structure definitions, some of which may represent interfaces to library routines that expect the default alignment rules. The `min_align` directive provides a way of limiting alignment changes to named structure and union types.

The syntax of the `min_align` directive is:

```
#pragma min_align {struct | union} tag n
```

where *tag* is the class, structure, or union tag to use and *n* is the minimum byte alignment to use for the structure. The alignment value, *n*, must be between 0 and 31. `Pragma min_align` must be used before the class, structure, or union definition begins, although it may be used after a forward reference of the class, structure. or union tag. As a side effect, the pragma introduces a forward reference to the named class, structure, or union.

This pragma specifies the minimum alignment to use. If the class, structure, or union contains a member that requires a larger alignment, then that larger alignment is used, and a warning message is issued.

Pragma `min_align` overrides any `#pragma align` that may be in effect, unless *n* is 0; in that case the `min_align` does not have an effect and the default alignment rules are used.

Example:

```
/* Force a set of alignment rules for
 * a particular struct, where doubles are
 * aligned to a 4-byte boundary.
 */

#pragma min_align struct old 4
#pragma align int 2
struct old {
    /* fields */
};
#pragma align double 0  /* reset to default */
```

Miscellaneous Directives

#pragma once

The compiler recognizes `#pragma once`, which, when placed at the beginning of a header file, indicates that the file is written in such a way that including it several times has the same effect as including it once. Thus, if the compiler sees `#pragma once` at the start of a header file, it will skip over the header file if that header file is `#include`d again.

A typical idiom is to place an `#ifndef` guard around the body of the file, with a `#define` of the guard variable after the `#ifndef`:

```
#pragma once                // optional
#ifndef FILE_H
#define FILE_H
... body of the header file ...
#endif
```

The `#pragma once` is marked as optional in this example, because the compiler recognizes the `#ifndef` idiom and does the optimization even in its absence. `#pragma once` is accepted for compatibility with other compilers and to allow the programmer to use other guard-code idioms.

#pragma ident

`#pragma ident` is recognized, in the same fashion as `#ident`:

```
#pragma ident string
#ident string
```

The compiler ignores these directives.

#pragma weak

The `weak pragma's` form is

```
#pragma weak name1 [ = name2 ]
```

where *name1* is the name to be given "weak binding" and is a synonym for *name2* if the latter is specified. See the "Executable and Linking Format (ELF)" chapter of the

Compilation Systems Volume 2 (*Concepts*) manual for more information on weak symbols.

Template Instantiation Pragmas

These pragmas provide explicit control over template instantiation. See Section “Instantiation #pragma Directives” on page 6-8 for more information.

Precompiled Headers

It is often desirable to avoid recompiling a set of header files, especially when they introduce many lines of code and the primary source files that `#include` them are relatively small. The EDG front end provides a mechanism for, in effect, taking a snapshot of the state of the compilation at a particular point and writing it to a disk file before completing the compilation; then, when recompiling the same source file or compiling another file with the same set of header files, it can recognize the “snapshot point,” verify that the corresponding precompiled header (“PCH”) file is reusable, and read it back in. Under the right circumstances, this can produce a dramatic improvement in compilation time; the trade-off is that PCH files can take a lot of disk space.

Automatic Precompiled Header Processing

When **`--pch`** appears on the command line, automatic precompiled header processing is enabled. This means the front end will automatically look for a qualifying precompiled header file to read in and/or will create one for use on a subsequent compilation.

The PCH file will contain a snapshot of all the code preceding the “header stop” point. The header stop point is typically the first token in the primary source file that does not belong to a preprocessing directive, but it can also be specified directly by `#pragma hdrstop` (see below) if that comes first. For example:

```
#include "xxx.h"
#include "yyy.h"
int i;
```

The header stop point is `int` (the first non-preprocessor token) and the PCH file will contain a snapshot reflecting the inclusion of **`xxx.h`** and **`yyy.h`**. If the first non-preprocessor token or the `#pragma hdrstop` appears within a `#if` block, the header stop point is the outermost enclosing `#if`. To illustrate, here’s a more complicated example:

```
#include "xxx.h"
#ifndef YYY_H
#define YYY_H 1
#include "yyy.h"
#endif
#if TEST
   int i;
#endif
```

Here, the first token that does not belong to a preprocessing directive is again `int`, but the header stop point is the start of the `#if` block containing it. The PCH file will reflect the inclusion of **`xxx.h`** and conditionally the definition of `YYY_H` and inclusion of **`yyy.h`**; it will not contain the state produced by `#if TEST`.

A PCH file will be produced only if the header stop point and the code preceding it (mainly, the header files themselves) meet certain requirements:

- The header stop point must appear at file scope — it may not be within an unclosed scope established by a header file. For example, a PCH file will not be created in this case:

  ```
  // xxx.h
  class A {
  // xxx.C
  #include "xxx.h"
  int i; };
  ```

- The header stop point may not be inside a declaration started within a header file, nor (in C++) may it be part of a declaration list of a linkage specification. For example, in the following case the header stop point is `int`, but since it is not the start of a new declaration, no PCH file will be created:

  ```
  // yyy.h
  static
  // yyy.C
  #include "yyy.h"
  int i;
  ```

- Similarly, the header stop point may not be inside a `#if` block or a `#define` started within a header file.
- The processing preceding the header stop must not have produced any errors. (Note: warnings and other diagnostics will not be reproduced when the PCH file is reused.)
- No references to predefined macros `__DATE__` or `__TIME__` may have appeared.
- No use of the `#line` preprocessing directive may have appeared.
- `#pragma no_pch` (see below) must not have appeared.
- The code preceding the header stop point must have introduced a sufficient number of declarations to justify the overhead associated with precompiled headers. The threshold is currently set to 1.

When a precompiled header file is produced, it contains, in addition to the snapshot of the compiler state, some information that can be checked to determine under what circumstances it can be reused. This includes:

- The compiler version, including the date and time the compiler was built.
- The current directory (i.e., the directory in which the compilation is occurring).
- The command line options.
- The initial sequence of preprocessing directives from the primary source file, including `#include` directives.
- The date and time of the header files specified in `#include` directives.

This information comprises the PCH "prefix." The prefix information of a given source file can be compared to the prefix information of a PCH file to determine whether the latter is applicable to the current compilation.

As an illustration, consider two source files:

```
// a.C
#include "xxx.h"
...                 // Start of code
// b.C
#include "xxx.h"
...                 // Start of code
```

When **`a.C`** is compiled with **`--pch`**, a precompiled header file named **`a.pch`** is created. Then, when **`b.C`** is compiled (or when **`a.C`** is recompiled), the prefix section of **`a.pch`** is read in for comparison with the current source file. If the command line options are identical, if **`xxx.h`** has not been modified, and so forth, then, instead of opening **`xxx.h`** and processing it line by line, the front end reads in the rest of **`a.pch`** and thereby establishes the state for the rest of the compilation.

It may be that more than one PCH file is applicable to a given compilation. If so, the largest (i.e., the one representing the most preprocessing directives from the primary source file) is used. For instance, consider a primary source file that begins with

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

If there is one PCH file for **`xxx.h`** and a second for **`xxx.h`** *and* **`yyy.h`**, the latter will be selected (assuming both are applicable to the current compilation). Moreover, after the PCH file for the first two headers is read in and the third is compiled, a new PCH file for all three headers may be created.

When a precompiled header file is created, it takes the name of the primary source file, with the suffix replaced by a suffix (`pch` by default). Unless **`--pch_dir`** is specified (see below), it is created in the directory of the primary source file.

When a precompiled header file is created or used, a message such as

```
    "test.C": creating precompiled header file "test.pch"
```

is issued. The user may suppress the message by using the command-line option **`--no_pch_messages`**.

When the **`--pch_verbose`** option is used the front end will display a message for each precompiled header file that is considered that cannot be used giving the reason that it cannot be used.

In automatic mode (i.e., when **`--pch`** is used) the front end will deem a precompiled header file obsolete and delete it under the following circumstances:

- if the precompiled header file is based on at least one out-of-date header file but is otherwise applicable for the current compilation; or
- if the precompiled header file has the same base name as the source file being compiled (e.g., **`xxx.pch`** and **`xxx.C`**) but is not applicable for the current compilation (e.g., because of different command-line options).

This handles some common cases; other PCH file clean-up must be dealt with by other means (e.g., by the user).

Support for precompiled header processing is not available when multiple source files are specified in a single compilation: an error will be issued and the compilation aborted if the command line includes a request for precompiled header processing and specifies more than one primary source file.

Manual Precompiled Header Processing

Command-line option **`--create_pch`** *file-name* specifies that a precompiled header file of the specified name should be created.

Command-line option **`--use_pch`** *file-name* specifies that the indicated precompiled header file should be used for this compilation; if it is invalid (i.e., if its prefix does not match the prefix for the current primary source file), a warning will be issued and the PCH file will not be used.

When either of these options is used in conjunction with **`--pch_dir`**, the indicated file name (which may be a path name) is tacked on to the directory name, unless the file name is an absolute path name.

The **`--create_pch`**, **`--use_pch`**, and **`--pch`** options may not be used together. If more than one of these options is specified, only the last one will apply. Nevertheless, most of the description of automatic PCH processing applies to one or the other of these modes — header stop points are determined the same way, PCH file applicability is determined the same way, and so forth.

Other Ways for Users to Control Precompiled Headers

There are several ways in which the user can control and/or tune how precompiled headers are created and used.

- `#pragma hdrstop` may be inserted in the primary source file at a point prior to the first token that does not belong to a preprocessing directive. It enables the user to specify where the set of header files subject to precompilation ends. For example,

  ```
  #include "xxx.h"
  #include "yyy.h"
  #pragma hdrstop
  #include "zzz.h"
  ```

  Here, the precompiled header file will include processing state for **`xxx.h`** and **`yyy.h`** but not **`zzz.h`**. (This is useful if the user decides that the information added by what follows the `#pragma hdrstop` does not justify the creation of another PCH file.)

- `#pragma no_pch` may be used to suppress precompiled header processing for a given source file.

- Command-line option **`--pch_dir`** *directory-name* is used to specify the directory in which to search for and/or create a PCH file.

Performance Issues

The relative overhead incurred in writing out and reading back in a precompiled header file is quite small for reasonably large header files.

In general, it doesn't cost much to write a precompiled header file out even if it does not end up being used, and if it *is* used it almost always produces a significant speedup in compilation. The problem is that the precompiled header files can be quite large (from a minimum of about 250K bytes to several megabytes or more), and so one probably doesn't want many of them sitting around.

Thus, despite the faster recompilation, precompiled header processing is not likely to be justified for an arbitrary set of files with nonuniform initial sequences of preprocessing directives. Rather, the greatest benefit occurs when a number of source files can share the same PCH file. The more sharing, the less disk space is consumed. With sharing, the disadvantage of large precompiled header files can be minimized, without giving up the advantage of a significant speedup in compilation times.

Consequently, to take full advantage of header file precompilation, users should expect to reorder the `#include` sections of their source files and/or to group `#include` directives within a commonly used header file.

The front end source provides an example of how this can be done. A common idiom is this:

```
#include "fe_common.h"
#pragma hdrstop
#include ...
```

where **`fe_common.h`** pulls in, directly and indirectly, a few dozen header files; the `#pragma hdrstop` is inserted to get better sharing with fewer PCH files. The PCH file produced for **`fe_common.h`** is a bit over a megabyte in size. Another idiom, used by the source files involved in declaration processing, is this:

```
#include "fe_common.h"
#include "decl_hdrs.h"
#pragma hdrstop
#include ...
```

`decl_hdrs.h` pulls in another dozen header files, and a second, somewhat larger, PCH file is created. In all, the fifty-odd source files of the front end share just six precompiled header files. If disk space were at a premium, one could decide to make **`fe_common.h`** pull in *all* the header files used — then, a single PCH file could be used in building the EDG front end.

Different environments and different projects will have different needs, but in general, users should be aware that making the best use of the precompiled header support will require some experimentation and probably some minor changes to source code.

Intrinsic Functions

Most *intrinsic functions* (intrinsics) provide access to hardware-related machine instructions. Others provide inline substitutes for some time-critical functions.When the **`-F`** option is used, the compiler recognizes certain function names as being intrinsic functions. To override this on any given use, enclose the name in parentheses. Intrinsic functions are used in the same manner as normal functions; however, rather than calling a subroutine, the compiler generates code directly. To use an intrinsic, it must be prototyped and marked as an intrinsic with a `pragma`. For example:

```
extern “C” int abs(int);
#pragma intrinsic abs
```

Any calls to a function declared like this will refer to the intrinsic.

For example, the following code, when compiled with the **`-F`** option, uses the `abs` intrinsic function:

```
extern “C” int abs(int);
#pragma intrinsic abs
sub(int arg)
{
   return abs(arg);
}
```

However, the following code does <u>not</u> use the intrinsic function:

```
extern int abs(int);
sub(int arg)
{
   return abs(arg);
}
```

The `abs` and `fabs` intrinsics return the absolute value of their argument. The syntax of these intrinsics follows:

```
int abs(x)
int x;

float fabs(x)
float x;

double fabs(x)
double x;
```

The `pow` and `powf` intrinsics handle certain common cases with inline code where the second operand is a small integral constant. Otherwise, they generate calls to `pow` or `powf`, as appropriate. See **`exp(3M)`**. The syntax of the `pow` and `powf` intrinsics follows:

```
double pow(x, y)
double x, y;

float powf(x, y)
float x, y;
```

The `_Test_and_Set` intrinsic can be used to generate an atomic test and set operation using machine-specific instructions that are not normally available for C++ programs.

```
int _Test_and_Set(pointer)
int *pointer;
```

Generates inline code to atomically set **pointer* to some unspecified non-zero value and returns the previous contents of **pointer*, under the assumption that `_Test_and_Set` is the only means by which **pointer* acquires a non-zero value.

The `__rot` intrinsic is used to rotate a word to the right.

```
unsigned int __rot(word, count)
unsigned int word;
int count;
```

Returns word, *word*, rotated to the right by *count* bits.

Refer to Chapter 3 “PowerPC Instruction Set Summary” of *Compation Systems Volume 1 (Tools)* (publication number 0890459) for a list of intrinsics for accessing specific PowerPC instructions. Refer to the Motorola document *AltiVec Technology Programming Interface Manual* for details regarding accessing AltiVec instructions.

AltiVec Technology Programming Interface

Motorola defined several extensions to C and C++ for accessing the AltiVec instructions of the PowerPC 7400 family of microprocessors. These extensions include new keywords, intrinsic functions, runtime functions, and a pragma. See the Motorola document *AltiVec Technology Programming Interface Manual* for detailed documentation of these extensions. This document will only discuss issues specific to Concurrent's implementation.

Currently the only microprocessor supported by Concurrent's C/C++ compiler that has AltiVec instruction is the MPC7400. It may be selected by the **`--target=ppc7400`** (or **`--target=mpc7400`**) option. See discussion of **`c.install`** and **`c.release`** commands for ways of setting the default target microprocessor if none is specified under the PLDE.

New Keywords for AltiVec

The following table shows the keywords that were added for AltiVec support and the compiler options to enable and disable them (note that **`bool`** is present in standard C++).

Keyword	Option to Enable	Option to Disable
`vector`	**`--vectors`**	**`--no_vectors`**
`pixel`	**`--vectors`**	**`--no_vectors`**
`bool`	**`--bool`**	**`--no_bool`**
`__vector`		N/A
`__pixel`		N/A

The following table documents the default settings for those compiler options:

C & target microprocessor has AltiVec instructions	C & target microprocessor does not have AltiVec instructions	C++
`--no_vectors`	**`--no_vectors`**	**`--no_vectors`**
`--no_bool`	**`--no_bool`**	**`--bool`**

New Intrinsic Functions for AltiVec

To access the intrinsic functions for AltiVec, include the header file **`<altivec.h>`**.

New Pragma for AltiVec

```
#pragma altivec_vrsave { on | off | allon | allzero }
```

The `allzero` option will set the VRSAVE register to zero in the procedure in which it is used. Refer to Motorola documentation for the other options.

varargs/stdarg for AltiVec

The base 5.3 release of Concurrent C/C++ has support **`varargs`**/**`stdarg`** for vector types. However, PowerMAX OS release 5.1 or later header files and libraries are required to use this feature.

Runtime for AltiVec

The Motorola defined `vec_malloc()`, `vec_calloc()`, `vec_realloc()`, and `vec_free()` are not implemented in the 5.0 release of PowerMAX OS. The standard system allocation routines - `malloc()`, `calloc()`, `realloc()`, and `free()` respectively - return quadword aligned memory (except for `free()` obviously), and can be used instead. Release 5.1 of PowerMAX OS includes these functions.

Interoperability with Non-AltiVec for AltiVec

The global variable `__vectors_present` allows the user to determine at runtime whether the system on which the program is running supports the AltiVec instruction set. Attempting to execute an AltiVec instruction on a system that does not support it will result in an Illegal Instruction exception. Any procedure that contains AltiVec instructions may have additional AltiVec instructions generated in the routines prolog and epilog.

In order to provide both normal and AltiVec accelerated versions of a routine, the following code sequence is recommended:

```
extern int __vectors_present;

type vector_function(arguments) {
   // Vector implementation of function
   ...
}

type function(arguments) {
   if (__vectors_present) {
      return vector_function(arguments);
   }

   // Non-vector implementation of function
   ...
}
```

Now, by calling `function()`, if the program is running an an AltiVec-supporting system, the program will execute the AltiVec accelerated version of the function. But if the program is run on a system that doesn't support AltiVec instructions, the program will not execute any AltiVec instructions.

Environment Variables

The environment variable `USR_INCLUDE` can be set to a directory to be used instead of **`/usr/include`** on the standard include file search list. (Of course, this has no effect if the front end has been configured to have an empty "standard list" of include files.)

Diagnostic Messages

Diagnostic messages have an associated *severity*, as follows:

- Catastrophic errors indicate problems of such severity that the compilation cannot continue. For example: command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.
- Errors indicate violations of the syntax or semantic rules of the C or C++ language. Compilation continues, but object code is not generated.
- Warnings indicate something valid but questionable. Compilation continues and object code is generated (if no errors are detected).
- Remarks indicate something that is valid and probably intended, but which a careful programmer may want to check. These diagnostics are not issued by default. Compilation continues and object code is generated (if no errors are detected).

Diagnostics are written to **`stderr`** with a form like the following:

```
"test.c", line 5: a break statement may only be used within a loop or switch
    break;
    ^
```

Note that the message identifies the file and line involved, and that the source line itself (with position indicated by the `^`) follows the message. If there are several diagnostics in one source line, each diagnostic will have the form above, with the result that the text of the source line will be displayed several times, with an appropriate position each time.

Long messages are wrapped to additional lines when necessary.

The **`--display_error_number`** may be used to request that the error number be included in the diagnostic message. When displayed, the error number also indicates whether the error may have its severity overridden on the command line. If the severity may be overridden, the error number will include the suffix "**`-D`**" (for "discretionary"); otherwise no suffix will be present.

```
"test_name.c", line 7: error #64-D: declaration does not declare anything
  struct {};
  ^
"test_name.c", line 9: error #77: this declaration has no storage class or
          type specifier
  xxxxx;
  ^
```

Because an error is determined to be discretionary based on the error severity associated with a specific context, a given error may be discretionary in some cases and not in others.

For some messages, a list of entities is useful; they are listed following the initial error message:

```
"test.c", line 4: error: more than one instance of overloaded function "f"
matches the argument list:
            function "f(int)"
            function "f(float)"
            argument types are: (double)
    f(1.5);
    ^
```

In some cases, some additional context information is provided; specifically, such context information is useful when the front end issues a diagnostic while doing a template instantiation or while generating a constructor, destructor, or assignment operator function. For example:

```
"test.c", line 7: error: "A::A()" is inaccessible
  B x;
     ^
detected during implicit generation of "B::B()" at line 7
```

Without the context information, it is very hard to figure out what the error refers to.

It is possible to change the severity level of certain messages by using an appropriate option.

`--diag_suppress` *tag*,*tag*,**...**

suppress the message

`--diag_remark` *tag*,*tag*,**...**

issue a remark

`--diag_warning` *tag*`,`*tag*`,...`

issue a warning

`--diag_error` *tag*`,`*tag*`,...`

issue an error

where *tag* is the message number (e.g., `0001`) or the message mnemonic name (e.g., `last_line_incomplete`).

For some messages, a list of entities is useful; they are listed following the initial error message:

```
“test.c”, line 4: error: more than one instance of overloaded function “f”
matches the argument list:
   function “f(int)”
   function “f(float)”
   argument types are: (double)
   f(1.5);
   ^
```

Termination Messages

```
n errors detected in the compilation of "ifile".
1 catastrophic error detected in the compilation of "ifile".
n errors and 1 catastrophic error detected in the compilation of "ifile".
```

is written to indicate the detection of errors in the compilation. No message is written if no errors were detected. The following message

`Error limit reached.`

is written when the count of errors reaches the error limit (see the **`-e`** option, above); compilation is then terminated. The message

`Compilation terminated.`

is written at the end of a compilation that was prematurely terminated because of a catastrophic error. The message

`Compilation aborted.`

is written at the end of a compilation that was prematurely terminated because of an internal error. Such an error indicates an internal problem in the compiler and should be reported to those responsible for its maintenance.

Response to Signals

The signals `SIGINT` (caused by a user interrupt, like `^C`) and `SIGTERM` (caused by a `kill` command) are trapped by the front end and cause abnormal termination.

Exit Status

On completion, the front end returns with a code indicating the highest-severity diagnostic detected: 4 if there was a catastrophic error, 2 if there were any errors, or 0 if there were any warnings or remarks or if there were no diagnostics of any kind.

If multiple source files are compiled, the exit status indicates the highest-severity diagnostic detected in the entire compilation.

Finding Include Files

A file name specified in a `#include` directive is searched for in a set of directories specified by command-line options and environment variables. If the file name specified does not include a suffix, a set of suffixes is used when searching for the file.

Files whose names are not absolute pathnames and that are enclosed in `"..."` will be searched for in the following directories, in the order listed:

1. The directory containing the current input file (the primary source file or the file containing the `#include`);[6]
2. any directories specified in **`--include_directory`** options (in the order in which they were listed on the command line);
3. any directories on the standard list (**`/usr/include`**).

For file names enclosed in `<...>`, only the directories that are specified using the **`--include_directory`** option and those on the standard list are searched. If the directory name is specified as "**`-`**", e.g., "**`-I-`**", the option indicates the point in the list of **`--include_directory`** options at which the search for file names enclosed in `<...>` should begin. That is, the search for `<...>` names should only consider directories named in **`--include_directory`** options following the **`-I-`**, and the directories of item 3 above. **`-I-`** also removes the directory containing the current input file (item 1 above) from the search path for file names enclosed in `"..."`.

An include directory specified with the **`--sys_include`** option is considered a "system" include directory. Warnings are suppressed when processing files found in system

6. However, if `STACK_REFERENCED_INCLUDE_DIRECTORIES` is TRUE, the directories of all the source input files currently in use are searched, in reverse order of `#include` nesting.

include directories. If a default include directory has been specified using the environment variable, it is considered a system include directory.

If the file name has no suffix it will be searched for by appending each of a set of include file suffixes. When searching in a given directory all of the suffixes are tried in that directory before moving on to the next search directory. The default set of suffixes is none, `.h`, and `.hpp`. The default can be overridden using the **`--incl_suffixes`** command-line option. A null file suffix cannot be used unless it is present in the suffix list (i.e., the front end will always attempt to add a suffix from the suffix list when the file name has no suffix).

7
Compilation Modes

7 Compilation Modes

This chapter describes the compilation modes of the Concurrent C++ compiler, **ec++** and the Concurrent C compiler, **ec**. Compiler options let you select a compilation mode; the modes are differentiated by their support for various dialects of C and C++.

Compilation modes are selected via compiler options. They are distinguished by their level of conformance to the ANSI C++ and ANSI C standards and dialects of C and C++. Table 7-1 depicts these modes and their command-line options.

Table 7-1. C++ Compilation Modes

cc++ Options	Mode	Description
(none)	Normal C++ mode	Accept normal ANSI C++ code.
--strict	ANSI C++ strictly-conforming mode	Compile any ANSI C++ strictly-conforming program.
--strict_warning	ANSI C++ stricting-conforming mode (warn on non-conforming usage)	Like **--strict**, but generates warning instead of error.
--cfront_2.1	**cfront** 2.1 compatibility mode	Support compatibility with **cfront** 2.1.
--cfront_3.0	**cfront** 3.0 compatibility mode	Support compatibility with **cfront** 3.0.
--anachronisms	Anachronism mode	Support various anachronisms ("Anachronisms Accepted" on page 5-5)

From least to most conforming, these compilation modes for C are: Old, Transition, ANSI C, and ANSI C Conforming. Table 7-1 depicts these modes and their command-line options.

Table 7-2. C Compilation Modes

cc++ Options	Mode Name	Description
-Xo	Old mode	Compatibility mode for pre-ANSI C code.

From least to most conforming, these compilation modes for C are: Old, Transition, ANSI C, and ANSI C Conforming. Table 7-1 depicts these modes and their command-line options.

Table 7-2. C Compilation Modes

cc++ Options	Mode Name	Description
-Xt	Transition mode	Use ANSI C semantics, but provide additional warning messages where those semantics conflict with previous practice.
-Xa	ANSI C mode	Silently use ANSI C features. This is the default compilation mode.
-Xc	(ANSI C) Conforming mode	Enforce ANSI C name-space restrictions and compile any strictly-conforming program.

See the **cc++(1)** man page for a brief description of these options and compilation modes. See Chapter 1 (“Compilation”) for more information on compilation.

When it is unimportant to distinguish among the Transition, ANSI C, and ANSI C Conforming compilation modes, the text refers to them collectively as the ANSI C compilation modes. Later sections describe each compilation mode and the features that characterize it. Table 7-2 summarizes the features that characterize the compilation modes.

ANSI C Mode

ANSI C mode provides a nearly standard ANSI C compilation environment. All ANSI C language syntax and semantics are obeyed. This mode differs from an ANSI C-conforming compilation environment because it supports the full system name-space in the header files; this allows access to all system macros and library routines even where forbidden for a strictly conforming ANSI implementation.

- All violations of ANSI C constraints on pointer type usage are diagnosed with warning messages.
- All system macros and library routines are available even where forbidden for a strictly conforming ANSI C implementation. For more information on name-space restrictions, see “Name-Space Restriction” on page 7-16.

Normal C++ Mode

This mode provides compatibility with the dialect of C++ described in the ARM. See Section “Extensions Accepted in Normal C++ Mode” on page 5-6 for more information.

Strictly-Conforming Mode

Strictly-conforming mode provides a compilation environment conforming to the emerging ANSI C++ standard. All ANSI C name-space restrictions are enforced, and all violations of ANSI C++ constraints and syntax rules produce warnings. This means that many variable names, function names, and macro names are not defined in this mode. To compile in strictly-conforming mode, invoke **`cc++`** with the **`--strict`** option. Strictly-conforming mode has the following characteristics:

Conforming mode is one of the three ANSI C compilation modes. This mode provides a compilation environment conforming to ANSI C. It is identical to ANSI C mode except that all ANSI C name-space restrictions are enforced, and all violations of ANSI C constraints and syntax rules produce warnings. This means that many variable names, function names, and macro names are not defined in this mode. To compile in Conforming mode, invoke **`hc`** with the **`-Xc`** option. Aside from those features listed in Table 7-2, Conforming mode has the following characteristics:

- The compiler does not predefine macros that violate the ANSI C++ and ANSI C name-space restrictions. Specifically, the predefined macros shown below are <u>not</u> available:

  ```
  unix
  _unix
  ```

- The following additional compiler error messages are generated for minor violations of ANSI C++ and ANSI C syntax semantic constraints:
  - Warnings for the use of an extra comma at the end of `enum` type declaration lists.
  - Warnings for missing declaration specifiers in external definitions.
  - Warnings for missing semicolons for the last item in `struct` and `union` declaration list.
  - Warnings for empty `struct` or `union` field declarations.
  - Warnings for anonymous bit fields with empty type specifiers.
  - Warnings for named bit fields with a size of zero.
  - Warning messages for empty (no tokens after preprocessing) source files.
  - Fatal errors for the use of the `$` character in identifier names. (A fatal error message is one for which **`ec++`** stops the compilation before producing an object file and returns an error code to its parent process.)
  - Fatal errors for the use of the `sizeof` on a bit-field. Normally **`cc++`** returns the size of the type of the bit-field.
  - Fatal error messages for using the `&` on variables declared with the register storage class. Normally **`ec++`** just produces a warning message.

`cfront` 2.1 Compatibility Mode

This mode provides compatibility with the **`cfront`** 2.1 dialect of C++. See “Extensions Accepted in cfront 2.1 Compatibility Mode” on page 5-8 for more information.

cfront 3.0 Compatibility Mode

This mode provides compatibility with the **`cfront`** 3.0 dialect of C++. See “Extensions Accepted in cfront 2.1 and 3.0 Compatibility Mode” on page 5-9 for more information.

Transition Mode

Transition mode is one of the three ANSI C compilation modes. It is intended to help customers convert their existing C code so it makes use of ANSI C semantics. This mode provides all the features of ANSI C mode, but the compiler issues additional diagnostic messages where ANSI C semantics conflict with the traditional semantics used in Old mode. To compile in Transition mode, invoke **`cc++`** with the **`-Xt`** option.

There are some cases where Transition mode cannot provide good diagnostics.

- In Transition mode, the ANSI C preprocessor attempts to produce diagnostics messages for **`cpp(1)`** preprocessing tricks but is unable to splice tokens at the beginning or end of macros. For more information on token-splicing, see “Preprocessing” on page 7-7 .
- Other undocumented features of **`cpp(1)`** may also fail to receive diagnostic messages.

  The illegal uses of type specifiers with typedefs generates syntax errors, instead of cleaner diagnostic messages. For example,

  ```
  foo()
  {
     typedef int integer ;
     {
        short integer local ;
     }
  }
  ```

  generates the error messages

  ```
  "file.c", line 5: syntax: deleting, ' local '
  "file.c", line 5: syntax: inserting, ' , IDENTIFIER' prior to ' ; '
  ```

  when compiled in any of the ANSI C compilation modes.

- Transition mode attempts to generate warning messages for code whose behavior may have changed because of changes in the default type-promotion rules or because of changes in the default types of literals. These messages do not always indicate a problem with the source code. Changes in default type-promotion may offset a change in the type of a literal or vice versa. The compiler only detects that at some point in an expression at least one difference in type-promotion occurred. For example, given the code

```
#define FLAG 0xffffffff
extern unsigned short us ;
foo()
{
   if (us + 1  < FLAG)
      return 1 ; /* always does this */
   return 0 ;
}
```

Transition mode generates the warning message

```
"file.c", line 4: warning: ANSI: Possibly
   different type-promotion around "<"
   operation
```

because under the value-preserving rules the expression

```
us + 1
```

has type `int` instead of `unsigned int` and because the literal

```
0xffffffff
```

has type `unsigned int` instead of `int`. Note, however, that the relational operator, **<** is always performing an unsigned comparison under either set of type-promotion rules.

Old Mode

Old mode provides maximal support for existing customer code and **`makefiles`**. However, there have been some compiler changes that may require source changes even in Old mode. These changes are discussed in “Library Enhancements” on page 7-16 .

In addition to the information provided in Table 7-1 , Old mode has the following feature:

- Some instances of illegal pointer usage receive caution messages rather than warning messages. The printing of caution messages must be explicitly enabled by using the **`-n`** option or the `#pragma cautions` directive.

Mode Features

Each compilation mode has features that characterize it. These features are described in the following sections.

Common Features

The following ANSI C features are available in all compilation modes.

- The function prototype syntax is always available. The PowerMAX OS header files, however, only use function prototype declarations if one of the ANSI C compilation modes (see “Compilation Modes” on page 7-1) is used. Please refer to Chapter 9 of *C: A Reference Manual* by Harbison and Steele and “Function Prototypes” on page 7-15 in this manual for more information about function prototypes.
- Initializers for automatic aggregate data objects such as structures and arrays are accepted in all compilation modes.
- Unions may be initialized. ANSI C defines the initializer for a union to initialize its first element.
- File-scoped declarations that use the `extern` storage class may be initialized, thus producing a definition.
- The `long double` data type is supported. As currently implemented, `long double` objects are of the same size and data format as objects of type `double`, however, future releases may use an extended-precision implementation for `long double`.
- Integer constants are permitted to use the “unsigned” suffixes `'u'` and `'U'`. For example, an integer literal of the form 123u or 123U has the type `unsigned int` rather than type `int`. Note: the `u` and `U` suffixes did not exist before ANSI C.
- Floating-point constants accept the suffixes `'f'` or `'F'`, and `'l'` or `'L'` to indicate `float` type or `long double` type constants, respectively. For example, 3.1415f and 3.1415F both produce constants with type `float`, while 3.1415l and 3.1415L produce constants with type `long double`.
- String constants that are separated by only white space are automatically concatenated. For example:

  ```
  "hello" " world\n"
  ```

  is treated as

  ```
  "hello world\n"
  ```

  rather than as a syntax error.
- The `signed`, `const`, and `volatile` keywords are always recognized by the compiler in Old mode.

- The syntax and library routine support for wide and multi-byte character constants and strings is available. Currently no wide or multi-byte locales are supported.

Differentiating Features

The C language defined by the ANSI C standard has a number of incompatibilities with older dialects of the language. This section attempts to explain the major areas of incompatibility introduced by the ANSI C standard that are likely to have significant impact on existing source code.

Preprocessing

Older releases of the C compiler used an internal preprocessor. Now, in Old mode the compiler uses the traditional UNIX preprocessor **`cpp(1)`**. In either case, ANSI C-specific preprocessing features are not available.

The ANSI C compilation modes use the **`/usr/ccs/lib/acpp`** preprocessor. This is a separate standalone preprocessor written to the ANSI C standard. (This processor is not considered to be a separately supported tool, but rather as a part of the compiler. A man page is not provided, and future releases of the C compiler may not provide it as a separate tool. Customers should not introduce any dependencies on its existence or undocumented aspects of its behavior.)

Providing the first complete specification of the C preprocessor is the ANSI C standard's most significant change. This change introduced a number of incompatibilities with the preprocessing supported by the traditional UNIX preprocessor **`cpp(1)`** and with the internal preprocessor used by older releases of the C compiler. These incompatibilities are:

- Trigraphs are sequences of three adjacent characters that are mapped into single characters during the first translation phase (before string constants and comments have been recognized). These sequences are provided to support non-ASCII hardware environments where certain common C characters are not available. The trigraphs are shown in Table 7-3 .

  The trigraph sequences are seldom encountered in non-ANSI C source; however, they do introduce a potential incompatibility for some uses in string constants, character constants, or in `#include` header file names. For example, the following statement appears differently in Old mode and in ANSI C mode after trigraph substitutions:

  Old mode:

  ```
  printf("warning: very strange error condition??!\n");
  ```

  ANSI C mode:

  ```
  printf("warning: very strange error condition|\n");
  ```

Table 7-3. Trigraph Mapping

Trigraph	Corresponding Character
??=	#
??(	[
??)	]
??/	\
??’	^
??<	{
??>	}
??!	|
??-	~

- ANSI C introduced an explicit token splicing operator, ##, that causes adjacent tokens in macros (`#defines`) to be spliced into a single token. ANSI C also reinforces the original Kernighan and Ritchie rule that specifies that comments are replaced with white space. This makes the traditional **`cpp(1)`** trick of using comment deletion in macros to perform token splicing illegal. For example, given a macro of the form

  ```
  #define declare_stack(name, type) \
  type *name/**/_STACK ;
  ```

  `cpp` interprets it as being equivalent to

  ```
  #define declare_stack(name, type) \
  type *name_STACK ;
  /* there is no space between the e and the _ */
  ```

  ANSI C interprets it as being equivalent to

  ```
  #define declare_stack(name, type) \
  type *name _STACK ;
  /* there is a space between the e and the _ */
  ```

  To get the intended token splicing behavior, use instead

  ```
  #define declare_stack(name, type) \
  type *name ## _STACK ;
  ```

  Other examples of **`cpp(1)`** token splicing that rely on splicing tokens at the beginning or end of macro expansions do not work under ANSI C preprocessing rules. For example, given

```
    #define DIGIT    5
    #define EPSILON  1.0e-DIGIT
```

an ANSI C preprocessor expands EPSILON to

```
    1.0e-DIGIT
```

However, because of the ANSI C definition of *preprocessing tokens*, **`cpp(1)`** expands EPSILON to

```
    1.0e-5
```

- The ANSI C specification also disallows the common practice of substituting macro parameters into string and character constants. For example, given

```
    #define str(x) "x\n"
    #define CNTRL(x) ('x' & 0x80)
    char *string = str(hello) ;
    char character = CNTRL(a) ;
```

an ANSI C preprocessor would produce

```
    char *string =  "x\n" ;
    char character =  ( 'x' & 0x80 ) ;
```

However, **`cpp(1)`** would produce

```
    char *string =  "hello\n" ;
    char character =  ('a' & 0x80) ;
```

To allow the creation of string constants from macro parameters ANSI C added a new preprocessing operator, # (sometimes called the *stringize* or *stringization* operator). It, together with the automatic concatenation of adjacent string constants, can be used to define the str() macro as:

```
    #define str(x)#x "\n"
```

The following macro call

```
    char *string = str(hello) ;
```

expands as follows, producing a valid ANSI C initializer

```
    char *string =  "hello" "\n" ;
```

Two approaches may be taken to handle macros that substitute parameters into character constants, like CNTRL() above.

The first method is to define the macro so that the argument must be contained within quotes. This requires that all uses of the macro be changed to put quotes around the arguments. For example,

```
#define CNTRL(x) (x & 0x80)
char    character = CNTRL('a') ;
```

This definition works with either ANSI C or old **`cpp(1)`** style preprocessors, but requires you to change all references to the macro in question.

The second method is to use the stringization operator and reference the first element of the string (recall that string constants are defined to be arrays of characters). For example, define the CNTRL macro as

```
#define CNTRL(x) (#x[0] & 0x80)
```

The following macro call

```
CNTRL(a)
```

expands to

```
( "a" [ 0 ] & 0x80 )
```

This expression has the same value as ('a' & 0x80). The principal advantage of this method is that it does not require you to change all references to the macro, but note that this form of the macro does not expand to a constant expression and so may not be used in data initializers or in case label expressions. This method also depends on the use of an ANSI C preprocessor.

Type-Promotion Rules

The ANSI C standard's most significant change to the semantics of the language is the introduction of new rules for type-promotion in expressions and changes in the default types of integer constants. This change can introduce undiagnosed changes in the behavior of programs that rely on the old type-promotion rules.

- Information about the handling of type-promotion in expressions follows:

  In Old mode, **`cc++`** (like many other C compilers) uses unsigned-preserving rules for type promotion in expressions. Under the unsigned-preserving rules, any expression that involves an unsigned type (`unsigned char`, `unsigned short`, `unsigned int`, or `unsigned long`) always promotes to an unsigned data type (`unsigned int` or `unsigned long`, as appropriate).

  ANSI C standardized the default type-promotion rules to follow what are called value-preserving semantics. Under the value-preserving rules, the smaller unsigned types (`unsigned char` and `unsigned short`) promote to the next larger type that can still represent all of the values of the unsigned data type. Note that this makes C's type-promotion rules dependent on the size of the data types in any particular implementation.

  For example, if a compiler's implementation of the `int` data type is large enough to represent all of the values representable in an `unsigned` `short` (i.e., `sizeof(unsigned short) < sizeof(int)`), then an expression that mixes an `unsigned short` with an `int` promotes to `int` type under the value-preserving rules, but `unsigned int` type under the unsigned-preserving rules. A different compiler that implemented `short` and `int` types with the same representation would promote that same expression to `unsigned int` type under the value-preserving rules.

For the current implementation of the cc++ compiler, the following relationships are true for all target machines:

- `sizeof(unsigned char) < sizeof(unsigned short)`
- `sizeof(unsigned short) < sizeof(int)`
- `sizeof (int) == sizeof(long)`

  This means that under the value-preserving rules, expressions involving `unsigned char` and `unsigned short` types promote to signed `int` type, instead of `unsigned int` as they would under the unsigned-preserving rules. Expressions promote to `unsigned int` types only if an `unsigned int` appeared explicitly in the expression. Likewise, expressions mixing `unsigned int` types with `long` types still promote to `unsigned long` under the value-preserving rules (because `sizeof(int) == sizeof(long)`, on all current targets).

- Information about the implicit data-typing of integer constants follows:

  ANSI C introduced new rules for determining the data types of integer constants (decimal, hex, and octal constants). In Old mode, these constants are always treated as having signed `int` or signed `long` (if the `L` suffix is used) type. You may apply the ANSI `'u'` or `'U'` unsigned suffix and force the literals to have `unsigned int` or `unsigned long` type (if the `L` suffix is also used).

  The ANSI C standard specifies that the data types of integer constants are chosen by picking the smallest type that can represent the constant value in the list below. Like the value-preserving type-promotion rules (discussed before), this rule depends on a particular implementation's representation of the `int` and `long` data types. In Table 7-4 parenthesized data types are valid for ANSI C but are redundant for cc++ because cc++ implements the `int` and `long` data types with the same 32-bit two's complement representation.

Table 7-4. Constants and Type Lists

Constant	Type List
unsuffixed decimal:	int, (long), unsigned long
unsuffixed hex or octal:	int, unsigned int, (long), (unsigned long)

Table 7-4. Constants and Type Lists (Cont.)

Constant	Type List
U suffixed:	unsigned int, (unsigned long)
L suffixed:	long, unsigned long
UL suffixed:	unsigned long

The information in Table 7-4 translates to the following facts. The net effect of applying the ANSI rules to the current target machines (where `sizeof (int) == sizeof (long)`) is that “unsuffixed” decimal literals that cannot be represented as a signed `int` have type `unsigned long`, and octal and hex literals that cannot be represented as a signed `int` have type `unsigned int`. Similarly, literals that use the `'l'` or `'L'` long suffixes automatically promote to `unsigned long` if they cannot be represented as a signed `long`. Literals that use `'u'` or `'U'` unsigned suffixes have `unsigned int` or `unsigned long` type (if the `L` suffix is also used).

Table 7-5 shows the difference in type rules for the numbers 1 and 4294967295. For cc++’s current target machines, the value 1 can be represented as a 32-bit, signed, sstwo’s complement integer constant but 4294967295 cannot be.

In the cases marked as overflowing, the old type-promotion rules force the constant’s value to overflow the representation. The result on current machines is that the constant receives a different numeric value (-1 instead of 4294967295). This can be the source of silent changes in behavior.

Table 7-5. Constant Representations

Constant	Non-ANSI C Type	ANSI C Type
1	int	int
4294967295	int (overflows)	unsigned long
0x1	int	int
0xffffffff	int (overflows)	unsigned int
1L	long	long
0xffffffffL	long (overflows)	unsigned long
1U	unsigned int	unsigned int
0xffffffffU	unsigned int	unsigned int
0x1UL	unsigned long	unsigned long
0xfffffffUL	unsigned long	nsigned long

Binary Operator Expressions

An expression that involves a binary operator is a binary expression. Before a binary expression is evaluated, the two operands may be converted. C is especially lenient in allowing mixed operands in expressions. Before evaluating most binary expressions, C converts all operands to a common data type. In Old mode, the exact sequence of conversion the compiler takes before evaluating an arithmetic expression is as follows:

- Any `signed char,short` or `signed short` operand is converted to `int`;
- Any `char,` `unsigned char` or `unsigned short` operand is converted to `unsigned int`;
- If one operand is `double`, the other is converted to `double` and the result type is `double`;
- Otherwise, if one operand is `float`, the other is converted to `float` and the result type is `float`;
- Otherwise, if one operand is `unsigned long`, the other is converted to `unsigned long` and the result type is `unsigned long`;
- Otherwise, if one operand is `long` and the other is `unsigned int`, then they are both converted to `unsigned long` and the result type is `unsigned long`;
- Otherwise, if one operand is `long`, the other is converted to `long` and the result type is `long`;
- Otherwise, if one operand is `unsigned int`, the other is converted to `unsigned int` and the result type is `unsigned int`;
- Otherwise, both operands are `int` and the result type is `int`.

  For example, in the following program, the same value is assigned to a `char`, `int`, and `float` variable:

```
main()
{
    float f;
    int i;
    char c;
    /* assign 'A' to c, 65 to i, 65.0 to f */
    f = i = c = 'A' ;
    printf("c=%c  i=%5d  f=%5.1f \n",c,i,f);
}
```

  The output is:

```
c = A   i = 65   f = 65.0
```

  The character 'A' is converted to an integer when assigned to i, then converted to a real number when assigned to f.

The following list is the subset of the ANSI C type-promotion scheme that is applicable in the ANSI C compilation modes of the Concurrent C compiler.

- Any `unsigned char`, `char`, `signed char`, `unsigned short`, `short`, or `signed short` is converted to `int`.
- If one operand is an `unsigned int` or `unsigned long`, any `int` (or value converted to `int`) is converted to `unsigned int`.
- If one operand is `double`, the other is converted to `double` and the result type is `double`.
- Otherwise, if one operand is `float`, the other is converted to `float` and the result type is `float`.

Escape Characters

ANSI C allows new escape characters in string and character literals. In older releases of the C compiler and in Old mode, **`cc++`** does not recognize these escape characters.

In the ANSI C compilation modes, the sequence `\a` is interpreted as the alert character and the sequence `\x` introduces a hexadecimal escape sequence (similar to octal escape sequences). (The actual encoding of the alert character is defined by ANSI C as being implementation-dependent. The **`cc++`** compiler implements it as the ASCII BEL character, so in this example, the equivalent octal escape code for BEL is shown.)

Prior to ANSI C the following two strings

```
 "\a1 is nothing special" "\x2 is nothing special"
```

would have been treated by the compiler as equivalent to the strings

```
"a1 is nothing special" "x2 is nothing special"
```

However, under the ANSI C standard, they are treated as being equivalent to

```
"\0071 is nothing special" "\02 is nothing special"
```

This can cause changes in programs that inadvertently make use of the `\a` or `\x` escape sequences.

Redeclaration of Typedefs

ANSI C decided that type identifiers (`typedefs`) may be redeclared as normal identifiers (or other type identifiers) in inner scopes. At the same time, ANSI C made the common practice of allowing the unsigned and signed type specifiers to be mixed with integer type identifiers illegal. For example, the following code:

```
typedef int integer ;
unsigned integer local ;
```

is considered to be illegal in ANSI C; however, older releases of the C compiler and Old mode accept it as declaring local to be an `unsigned int`. On the other hand, ANSI C treats

```
typedef int integer
int foo()
```

```
{
 unsigned integer ;
}
```

as redeclaring integer as an identifier with type `unsigned int`. Older releases of the C compiler consider this to be a syntax error. Current releases of **`cc++`** (in Old mode) accept this as an extension.

Scope of Parameters

In Old mode, the formal parameters of a function definition are given a different scope than the variables declared in the first block of a function. This makes it legal to redeclare the name of a formal parameter as a variable in the first block of a function.

In the ANSI C compilation modes, function parameters have the same scope as variables declared in the first block of the function. This makes code like

```
function (parameter)
int parameter ;
{
 short parameter ;/* redeclare a parameter */
}
```

illegal in ANSI C since there are multiple definitions of `parameter` in the same scope. Names of formal parameters may still be redeclared in subsequent nested blocks. In Old mode, this is legal but probably not intentional.

Header File Features

The header files under **`/usr/include`** provide the following support for ANSI C:

- Meet ANSI C requirements on their contents
- Allow you to take advantage of the additional compile-time error checking available with function prototypes

Function Prototypes

Function prototypes are available for functions declared in most system header files. This has two advantages:

- The compiler can do better error-checking on the type and number of arguments to library routine calls.
- The single-precision math library routines can be used (assuming `#include <math.h>` appears) without resorting to the **`-fsingle2`** command-line option to **`cc++`**.

Name-Space Restrictions

The system header files contain conditional compilation code (`#ifdef __STDC__`) that controls the enforcement of ANSI C restrictions on their contents. In Old mode, the `__STDC__` macro is undefined; this means that ANSI C syntax, semantics, and name-space restrictions are not enforced. In ANSI C conforming mode (**`-Xc`** option), the `__STDC__` macro is defined to be 1, causing enforcement of the ANSI C syntax, semantics, and name-space restrictions. In Transition mode and ANSI C mode, the `__STDC__` macro is defined to be 0 (zero), indicating use of ANSI C syntax and semantics but not the enforcement of ANSI C name-space restrictions.

A POSIX^{TM}-conforming name-space may be achieved in any compilation mode by defining the macro `_POSIX_SOURCE` before any header files are included (the POSIX 1003.1 name-space is a superset of ANSI C name-space).

In Figure 7-1, the User Name-Space represents the universe of identifiers, the ANSI C and C++ Name-Space represents the identifiers defined by the ANSI C and C++ standards, and the shaded region represents identifiers that are defined in ANSI C and C++ header files but that are not defined by the ANSI C and C++ standards. The shaded region is given to the user in Conforming mode but is given to the system in all other modes.

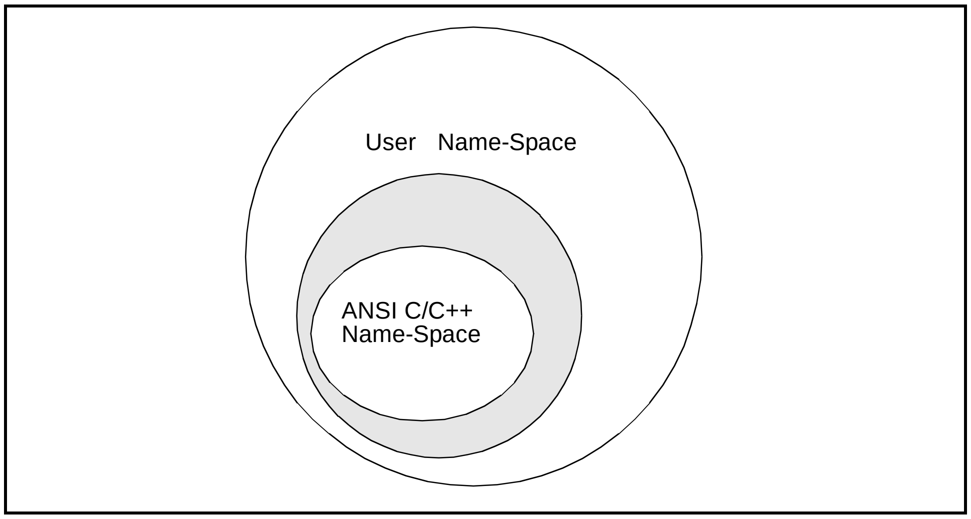

Figure 7-1. Name-Space Restriction

Library Enhancements

ANSI C specified a number of minor changes in the run-time behavior of several C library routines. Where ANSI changes conflict with existing practice, the library behavior is determined by the compilation mode specified on the command line when the final executable is <u>linked</u>. Care must be taken with **`makefiles`** (or other ad hoc **`make`** procedures) that do not automatically include compilation options with the link command or which create the final executable by using **`ld(1)`** directly.

The **cc++** driver program controls run-time behavior by linking special object files into the executable depending on the compilation mode. By default, executables are linked in the ANSI C mode. In the ANSI C (**-Xa**) and Transition (**-Xt**) modes, the object file **/usr/ccs/lib/ansi.o** is linked in to force ANSI C behavior. In the ANSI C conforming mode (**-Xc**), the object file **/usr/ccs/lib/strict.o** is used. In Old mode, no special object file is required.

For example, to compile and link a C program using the ANSI C compilation mode and run-time library behavior, use a command line like the one supplied here

```
$ ec -o prog file.c
```

To compile a C program using the ANSI C compilation mode but to link in Old mode, use a command line like the one supplied here

```
$ ec -c file.c; ec -Xo -o prog file.o
```

This might cause unexpected behavior if the code in **file.c** depends on ANSI C library behavior that conflicts with Old mode behavior. For more information on compilation, see Chapter 1 (“Compilation”)and the **cc++(1)** man page.

Note that **ld(1)** does not support the compilation mode options, so to force ANSI C behavior, either **/usr/ccs/lib/ansi.o** or **/usr/ccs/lib/strict.o** must be added to the link command. The **ld** command line to link an executable and force ANSI C behavior from the run-time library would be

```
$ ld -o prog /usr/ccs/lib/crt0.o /usr/ccs/lib/ansi.o foo.o -lc
```

The actual differences in library routine behavior are discussed in detail in the man pages for the modified routines. These are:

- **ctime(3C)**
- **exp(3M)**
- **frexp(3C)**
- **matherr(3M)**
- **printf(3S)**
- **scanf(3S)**
- **setbuf(3S)**
- **sinh(3M)**
- **strtol(3C)**
- **trig(3M)**

The major differences are in error-detection and handling. For example, in ANSI C mode the `strtol()` routine only accepts input in the range `LONG_MIN` to `LONG_MAX` and math library routines do not call the `matherr` routine if linked in ANSI C conforming mode (since ANSI C forbids this).

Locale-Support Enhancements

The following additional support is now provided by **cc++:**

- The `LC_COLLATE` locale category is now supported:
- A new tool, **`colltbl(1M)`**, is provided to define locale-dependent collating sequences.
- **`setlocale(3C)`** has been enhanced to support the `LC_COLLATE` category.
- Two new library routines, **`strxfrm(3C)`** and **`strcoll(3C)`**, are provided to support programs that wish to make use of locale-dependent collating sequence information.
- The following enhancements have been made to **`chrtbl(1M)`**
  - `chrtbl` now supports definition of the **`LC_NUMERIC`** (non-monetary numeric formatting information) locale data file. Previously, this data file had to be created manually. See **`localeconv(3C)`**.
  - `chrtbl` now contains preliminary support for the definition of multi-byte and wide-character locale definitions. This is done with the `cswidth chrtbl` specifier.

Multi-byte locales are not currently supported, and these features are unused.

Anachronism Mode

This mode supports various anachronisms from **`cfront`** dialects of C++. See “Anachronisms Accepted” on page 5-5 for more information.

8
Runtime Libraries

8
Runtime Libraries

This chapter identifies the libraries provided with the Concurrent C++ compilation system. This release of the compilation system includes runtime libraries that nearly in full compliance with *The ISO/IEC 14882:1998(E) C++ Standard*.

Previous releases included specialized C++ class libraries from another vendor. The core, standardized functionality in those libraries is now supported by the runtime library, and third-party libraries have been removed from the general release. They are available separately, however, for customers who rely heavily on the specialized functionality therein.

The set of C-compatible system runtime libraries provided in Concurrent's PowerMAX operating system, such as the math library and the networking libraries, may be included in programs compiled with the Concurrent C++ compilation system. The **`ec++`** driver always includes the standard C library among the libraries whose names are passed to the link editor.

The runtime library provided with the Concurrent C++ compilation system is:

- Runtime Library: **`libCruntime`**

In addition, the following third-party libraries are available from Concurrent separately:

- Cfront Libraries
  - I/O Library: **`libCio`**
  - Complex Library: **`libCcomplex`**

Runtime Library

General

The runtime library (**`libCruntime`**) includes Concurrent's implementation of the C++ Standard Library and the Language Support Library. A description of the Standard Library is beyond the scope of this manual, but there are many good references available at bookstores.

Language Support Library

The Language Support Library provides the following support:

- characteristics of predefined types

- program start and termination
- dynamic memory management
- dynamic type identification
- exception handling

Linking

The runtime library is provided in 4 forms: static (**`libCruntime.a`**), shared (**`libCruntime.so`**), thread-safe static (**`libCruntime_mt.a`**), and thread-safe shared (**`libCruntime_mt.so`**). The appropriate library is chosen based on the command line options used when invoking the compiler. These are summarized in Table 8-1.

Table 8-1. Choice of Runtime Library

		`-Zlink=static`	**`-Zlink=dynamic`** (default)
`threads`	**`-lthread`**	**`libCruntime_mt.a`** (thread-safe static)	**`libCruntime_mt.so`** (thread-safe shared)
	(default)	**`libCruntime.a`** (static)	**`libCruntime.so`** (shared)

Note that any sources that use the `<iostream>` header, and whose object files are to be linked with the -lthread option, should be compiled with the **`-D_REENTRANT`** option to specify thread safety.

Template Instantiation

A common source of confusion when using the runtime library is template instantiation. Certain templates, are instantiated in the library, either because they are required by the standard, they are needed by other parts of the library, or for convenience. Notably, many I/O template routines are instantiated for the types `char` and `wchar_t`. This is nice because simple programs can often be compiled without using any special command-line options or explicit instantiation in the source code. However, it can oversimplify the situation for user programs including template code.

When using template code with the C++ compiler, it is important to be familiar with the template instantiation command-line options. See “C++ Specific Features” on page 1-18 , “Template Instantiation” on page 6-3, and “Template Instantiation Pragmas” on page 6-28.

Cfront Libraries

In previous releases of the C++ compiler, Concurrent supplied a version of AT&T's cfront libraries. These libraries are no longer shipped with the compiler. They are available separately, but their use is discouraged. It is recommended that code using the older cfront libraries be migrated to the C++ Standard Library instead. Support for the cfront libraries will be dropped entirely in a future release.

I/O Library	The cfront I/O library (**`libCio`**) has been replaced by the I/O support in the Standard Library.
Complex Library	The cfront complex library (**`libCcomplex`**) has been replaced by the complex template class in the Standard Library.

If you insist on using the cfront libraries, you will need the **`--cfront_io`** command-line option which tells the compiler to use the old cfront I/O and which makes the compilation link with **`libCio`**.

When using the cfront complex library, you must explicitly tell the compiler to link with the **`libCcomplex`** by using the **`-lCcomplex`** option.

A
ANSI C++ Implementation

Although the ANSI C++ Working Paper defines many details of the C++ language, it leaves some areas to be defined by the implementation. This appendix explains how the Concurrent ANSI C++ implementation defines those areas. The appendix identifies each portion of the April 28, 1995, Working Paper which specifies an implementation-defined characteristic, along with the definition used in the Concurrent implementation.

Lexical Conventions (Chapter 2)

Phases of Translation (2.1)

Nonempty sequences of white-space characters are retained by the compiler.

Character Literals (2.9.2)

The value of a multicharacter literal that does not begin with the letter `L` is encoded into an integer character constant as follows:

```
    'a'     = 'a'
    'ab'    = 'a'<< 8 | 'b'
    'abc'   = 'a'<<16 | 'b'<< 8 | 'c'
    'abcd'  = 'a'<<24 | 'b'<<16 | 'c'<<8 | 'd'
```

Wide-character literals are only supported for single-character sequences. The value of a wide-character literal is the value of the right-most character enclosed in the single quotes.

If the value of the selected character in a character literal exceeds that of the largest `char` or `wchar_t`, the right-most byte of the selected character is regarded as the value of the character literal.

String Literals (2.9.4)

All string literals are distinct (that is, are stored in non-overlapping objects). The compiler **`-R`** option may be used, however, to pool string literals.

Basic Concepts (Chapter 3)

Types (3.9)

See Table 6-2, “Alignments by Data Type” for the alignment requirements of the various object types.

Main Function (3.6.1)

The type and the parameters of the `main` function are:

```
int main (int argc, char *argv[], char *envp[]);
```

The `main` function has external linkage.

Fundamental Types (3.9.1)

Table A-1 shows the sizes and value ranges of floating–point types. (The epsilon of a type is the difference between one and the next largest number that can be represented.) Table A-2 shows the sizes and value ranges of integer types

Table A-1. Floating-Point Types

Designation	Size (bits)	Range (decimal)
`float`	32	1.175494350822287e-38 through 3.40282346638528888540e+38 (The epsilon is 1.19209290e-07.)
`double`	64	2.22507385850720140e-308 through 1.79769313486231470e+308 (The epsilon is 2.77555756156289e-17.)
`long double`	64	2.22507385850720140e-308 through 1.79769313486231470e+308 (The epsilon is 2.77555756156289e-17.)

.

Table A-2. Integer Types

Designation	Size (bits)	Range (decimal)
`char`[a]	8	0 through 255 (any 8–bit unsigned integer)
`signed char`	8	-128 through 127 (ASCII characters plus negative bytes)
`unsigned char`	8	0 through 255 (any 8–bit unsigned integer)
`bool`	8	0 through 255 (any 8–bit unsigned integer)
`short`	16	-32768 through 32767
`signed short`	16	-32768 through 32767
`unsigned short`	16	0 through 65535
`int`	32	-2147483648 through 2147483647
`signed int`	32	-2147483648 through 2147483647
`unsigned int`	32	0 through 4294967295
`long`	32	-2147483648 through 2147483647
`signed long`	32	-2147483648 through 2147483647
`unsigned long`	32	0 through 4294967295
`wchar_t`	32	-2147483648 through 2147483647
`long long`	64	-9223372036854775808 through 9223372036854775807
`signed long long`	64	-9223372036854775808 though 9223372036854775807
`unsigned long long`	64	0 through 18446744073709551615[b]

a. A plain `char` object can take on the same values as a `signed char` if the **`--signed_chars`** option is used, or it can take on the same values as an `unsigned char` if the **`--unsigned_char`**s option is used. In the absence of either option, a plain `char` object takes on the same values as an `unsigned char`.
b. 18 quintillion, 446 quadrillion, 744 trillion, 73 billion, 709 million, 551 thousand, 615!

See Appendix B (“Architecture Dependencies”) for the value representations of floating-point and integer types

Standard Conversions (Chapter 4)

Integral Conversions (4.7)

When an integer type value is converted to a shorter integer type value, the original value is truncated, discarding the high-order bits which do not fit in the new type.

Expressions (Chapter 5)

Reinterpret Cast (5.2)

When a pointer is converted to an integral type, the mapping function is a conversion of the value of that pointer to the integral type, as if an explicit cast had been done to that integral type.

When an integral type is converted to a pointer, the mapping function is a conversion from the integral type to a value that can be represented as an `unsigned int`, as if an explicit cast had been done to `unsigned int`. The programmer must ensure that the value of the pointer represents a correct alignment of the type pointed to.

Sizeof (5.3.3)

`sizeof(bool)` is 1.

`sizeof(wchar_t)` is 4.

The result of the `sizeof` operator is a constant of type `unsigned int`.

Multiplicative Operators (5.6)

In the operation `E1 % E2`, the sign of the remainder is the sign of `E1`.

Additive Operators (5.7)

The result of subtraction of two pointers to elements of the same array object is of type `signed int`.

Shift Operators (5.8)

In the operation `E1 >> E2`, if `E1` is negative, the vacated bits of `E1` are one-filled.

Relational Operators (5.9)

Other pointer comparisons produce a result equivalent to that produced by a comparison of the pointer values each cast to type `unsigned int`.

Declarations (Chapter 7)

The asm declaration (7.4)

`asm()` is regarded as an ordinary function declaration. It is not used to provide inline assembly language code in a C program.

Linkage Specifications (7.5)

Only the linkage specifications “C” and “C++” are valid.

Linkage from C++ to objects defined in other languages, or from other languages to objects defined in C++, can be achieved by specifying

```
extern “C” {
}
```

around the declarations of the objects in the C++ code. Note that the objects’ link-level names, when used in languages other than C++, must be the same as the names specified in the linkage specification declaration. Linkage can also be achieved without use of the C linkage specification, provided that the objects’ link-level names, when used in languages other than C++, match the “mangled” names produced by the Concurrent C++ compiler.

Declarators (Chapter 8)

Default Arguments (8.3.6)

The order of evaluation of function arguments varies according to such factors as the context of the function call, the level of optimization used in compilation, etc.

Classes (Chapter 9)

Class Members (9.2)

Non-`static` data members separated by an access specifier are allocated within a class object in order of declaration. If member `y` is declared after member `x`, member `y` has a higher address than does member `x`.

Bit-fields (9.7)

Bit-fields are allocated from left to right (most to least significant bits). Bit-fields never cross over an alignment boundary for their type. However, multiple bit-fields which are sufficiently small may occupy the same allocation unit. For example, two `int` bit-fields whose total size is less than 32 bits may share a single 32–bit word.

However, if the first `int` bit-field is 17 bits and the second is 16 bits, there are 15 padding bits between them. Bit-fields may also share their allocation unit with other `struct` members. For example, a 16-bit `int` bit-field followed by a `short` occupies one 32–bit word.

A plain `int` bit-field is unsigned.

Special Member Functions (Chapter 12)

Temporary Objects (12.2)

The creation of temporaries by the compiler varies according to such factors as the context of the function call, the level of optimization used in compilation, etc.

Preprocessing Directives (Chapter 16)

Conditional Inclusion (16.1)

Since the source and destination character sets are identical, character constants have the same value whether they are in a preprocessing conditional statement or are in source code which is passed by the preprocessor to the compiler.

The above holds for the 7-bit ASCII characters. 8-bit characters are treated as unsigned by the preprocessor.

Source File Inclusion (16.2)

Includable source files whose names do not begin with "**/**" are searched for in the following manner:

If the name is enclosed in double-quotes (**""**), the file is searched for in the directory of the file containing the `#include` statement. If that search fails, or if the name is enclosed between a `<` and a `>`, the file is searched for under **`/usr/include`**. This behavior can be modified by using the **`-I`** command-line option. Refer to the **`cc++(1)`** man page for more details.

The name of the file to be included is the full name by which the file is known to the operation system. This may include an absolute or relative path. For example:

`<stdio.h>`	Refers to **`/usr/include/stdio.h`**
`<sys/time.h>`	Refers to **`/usr/include/sys/time.h`**
`"/usr/include/sys/time.h"`	Also refers to **`/usr/include/sys/time.h`**
`"fleas.h"`	Searches for **`fleas.h`** first in the directory where the including file is located, then in **`/usr/include`**
`"sys/fleas.h"`	Searches for **`fleas.h`** in the **`sys`** subdirectory (if any) of the directory in which the including file is located, then in **`/usr/include/sys.`**

Predefined Macro Names (16.8)

The date and time are always provided by the operating system. Therefore, no defaults exist for situations where the date and time of translation are not available.

See "Predefined Macros" on page 6-17 for the definition of `the __STDC__` macro.

Headers (Chapter 17)

Freestanding Implementations (17.3.1.3)

The implementation provided in the Concurrent C++ compilation system is hosted. A freestanding implementation is not provided.

Library Introduction (Chapter 17)

Reentrancy (17.3.4.5)

The following libraries are included:

- C system library: **`/usr/ccs/lib/libc.a`** and **`/usr/ccs/lib/libc.so`** provide reentrancy; **`/usr/ccs/lib/libnc.a`** does not.
- C++ runtime support library: **`/usr/ccs/lib/libCruntime.a`** does not provide reentrancy.
- C++ I/O support library: **`/usr/ccs/lib/libCio_mt.a`** provides reentrancy; **`/usr/ccs/lib/libCio.a`** does not.

Language Support Library

Class bad_alloc (18.4.2.1)

`what()` returns the empty character string (`““`).

Class bad_cast (18.5.2)

`what()` returns the empty character string (`““`).

Class bad_typeid (18.5.3)

`what()` returns the empty character string (`““`).

Class bad_exception (18.6.2.1)

`what()` returns the empty character string (`““`).

Class exception (18.6.1)

`what()` returns the empty character string (`““`).

Input/Output Library (Chapter 27)

Types (27.4.1)

The type `streamoff` is of type `long`.

The type `wstreamoff` is not currently supported in the Concurrent C++ compilation system.

The type `streampos` is of type `long`.

The type `wstreampos` is not currently supported in the Concurrent C++ compilation system.

basic_ios iostate flags functions (27.4.4.3)

The class `basic_ios::failure` is not currently supported in the Concurrent C++ compilation system.

Standard Manipulators (27.6.3)

The type `smanip` is not currently supported in the Concurrent C++ compilation system. The class `SMANIP` is supported. See **`manip(3c++)`** for more information.

Compatibility (Appendix C)

Predefined Names (16.8)

See “Predefined Macros” on page 6-17 for the definition of `the __STDC__` macro.

B
Architecture Dependencies

The PowerPC-based systems targeted by the Concurrent C/C++ compilation system are 32-bit word, two's complement computers. These systems support the following major data types: bit, byte, half-word, word, double-word, and floating-point.

Bit-Field

A *bit-field* is a structure member or union member that consists of 1 through 31 contiguous bits. Bit-fields may be of type `unsigned int`, `int`, and `signed int`. The compiler also allows them to be of types `unsigned`, `signed`, `char`, and `short`. Fields that are declared to be `unsigned int` are zero-extended to `unsigned int` type when used in an expression. Similarly, fields declared to be of type `signed int` are sign-extended to `int` type. Fields that are not explicitly declared to be signed or unsigned are zero-extended to `unsigned int` type. The use of bit-fields is often not portable.

The C/C++ compiler determines how bit-fields and structure members that take up less than a word are stored. Each of the following rules is applied before a member is stored.

- Members are packed in the order in which they were declared.
- Members are packed as tightly as possible.
- Members' data-alignment rules are followed.
- Members do not cross their storage unit boundaries; for example, if a field does not fit into the remaining space left in a word, it is placed into the next word. Fields declared as `char` or `short` behave just like int fields except that instead of word boundaries, they do not cross `char` and `short` boundaries, respectively.
- Unused space in storage units is padded.

Figure B-1 shows how the system stores sequentially defined bit-fields.

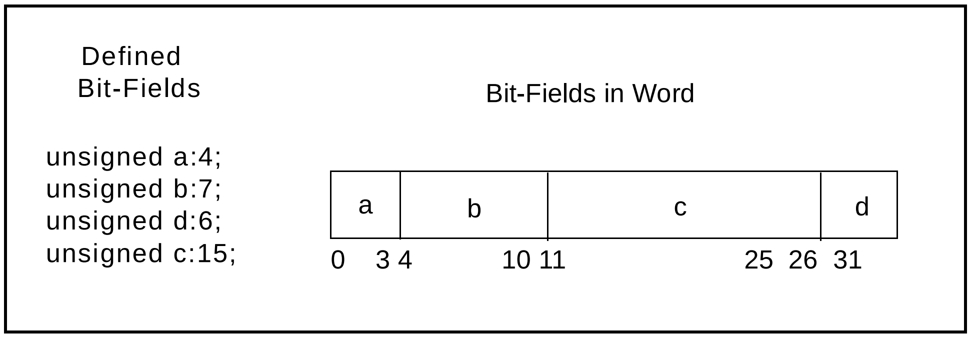

Figure B-1. Bit-Field Example

Structures may contain fields and members of other types and sizes. The size of a `struct` may not be equal to the sum of its members' sizes. This is because the alignment constraints of the individual members may force pad bits or bytes to be inserted between members, padding them to the next boundary appropriate for their declared type.

See "Data Alignment Rules" on page 6-25 for alignment constraints.

Byte

A *byte* contains eight bits starting on an addressable byte boundary. The most significant bit (MSB) designates the byte's address. Figure B-2 shows the address and MSB of a byte in the system.

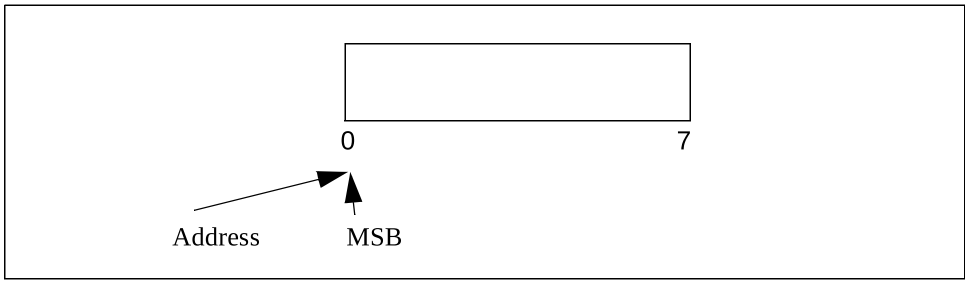

Figure B-2. Address and MSB of a Byte

If the byte is an unsigned integer, then its value is in the decimal range 0 - 255 (binary 00000000 - 11111111).

If the byte is a signed numeric integer, then it contains a two's complement value. As a two's complement number, a byte represents a decimal ranging from -128 to +127.

The following C/C++ data types take up one byte: `unsigned char`, `char`, `signed char`. The default for the Concurrent C/C++ compiler is to treat plain `char` variables as being unsigned.

Half-Word

A 16-bit *half-word* contains two bytes and starts on an addressable 16-bit word boundary. Figure B-3 shows that the MSB of the most significant byte is the half-word's address.

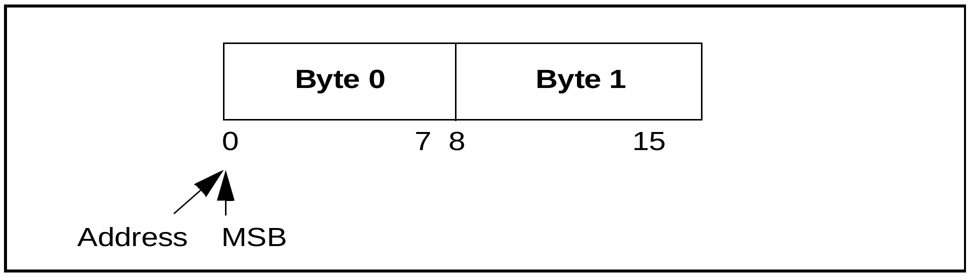

Figure B-3. Address and MSB of a Half-Word

If the half-word is an unsigned integer, then its decimal value ranges from 0 to 64K-1. If the half-word contains a signed numeric integer, then its two's complement value ranges from decimal -32K to +32K-1.

The following C/C++ data types take up one half-word: `unsigned short`, `short`, `signed short`.

Word

A *word* contains four bytes (32 contiguous bits). The word's address may be a word boundary or a CPU register. The MSB is the word's address. See Figure B-4. .

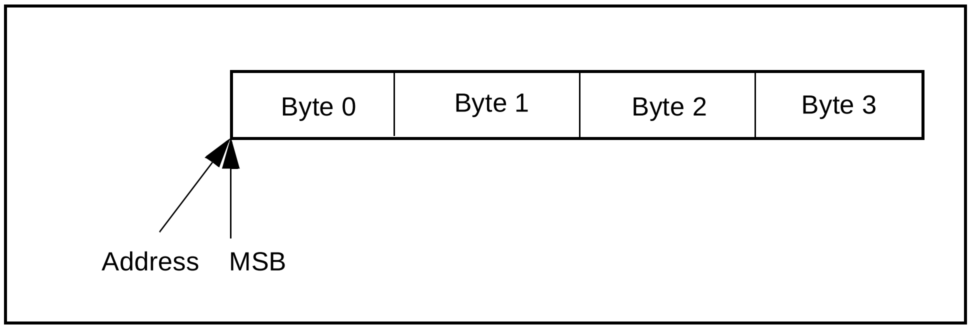

Figure B-4. Address and MSB of a Word

If the word is an unsigned integer, then its value ranges from decimal 0 to 2**32-1. As a signed numeric value, a word represents an integer from -2**31 to +2**31-1.

The following C/C++ data types take up one word: `unsigned int`, `int`, `signed int`, `unsigned long`, `long`, `signed long`. Enumerations are implemented as signed `ints`, and pointers are implemented as `unsigned ints`, so they also take up one word.

Conceptually, [`signed`] `int` and [`signed`] `long` represent different data types, where the size of a `long` is the same or larger than the size of an `int`. However, currently the PowerPC-based systems store both `ints` and `longs` in one 32-bit word of memory.

Double Word

A *double word* contains eight bytes (64 contiguous bits). The word's address may be a double word boundary or a CPU register. The MSB is the word's address. See Figure B-5 .

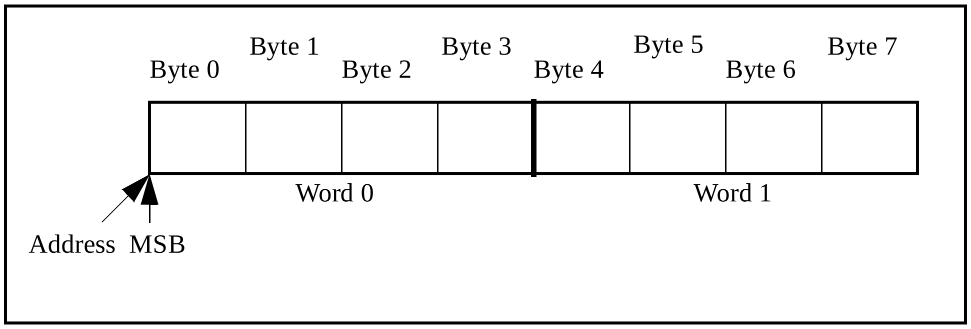

Figure B-5. Address and MSB of a Double Word

If the word is an unsigned integer, then its value ranges from decimal 0 to 2**64-1. As a signed numeric value, a word represents an integer from -2**63 to +2**63-1.

The following C/C++ data types take up two words: `unsigned long long int`, `long long int`, `signed long long int`, and `double`.

Shift Operations

The *shift operators* shift an integer by 0 through 31 bit positions to the left (<<) or to the right (>>).

```
shift-expression ::= e1 << e2
       e1 >> e2
```

The operands are the integer to be shifted (e1) and the number of bit positions by which it is to be shifted (e2). Both must be an integral type. The right operand, e2, must be within the range 0 through 31.

In a left shift, all 32 bits, including the sign bit, are shifted to the left, with zeros replacing the vacated rightmost bits.

A right shift has different effects depending on whether or not e1 is signed. If e1 is unsigned, then it is shifted e2 bits to the right, with zeros replacing the leftmost bits. If e1 is signed, however, then the sign bit replaces the vacated bits on the left.

Figure B-6 and Figure B-7 illustrate left and right shifts for both signed and unsigned quantities, where MSB is most significant bit and LSB is least significant bit.

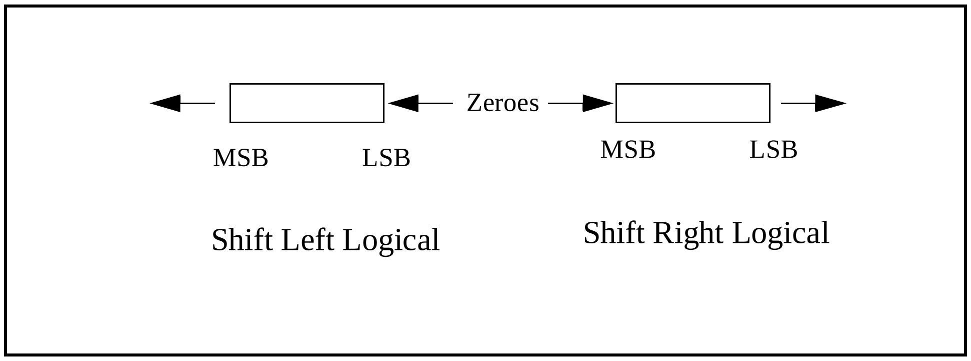

Figure B-6. Left/Right Shift of Unsigned Integer

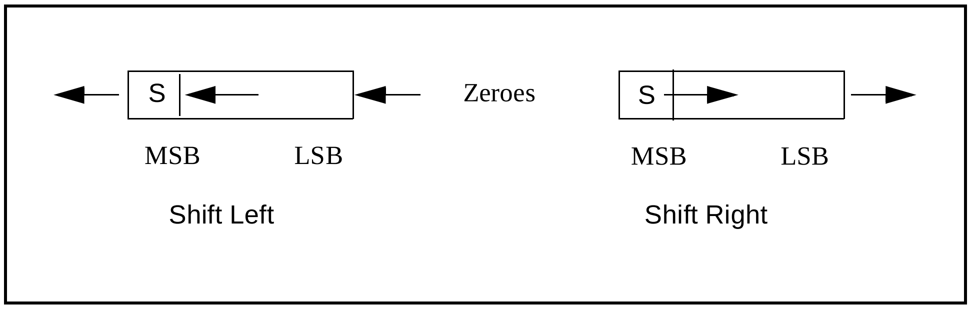

Figure B-7. Left/Right Shift of Signed Integer

Floating-Point

Figure B-8 shows the format of a 32-bit single-precision floating-point number.

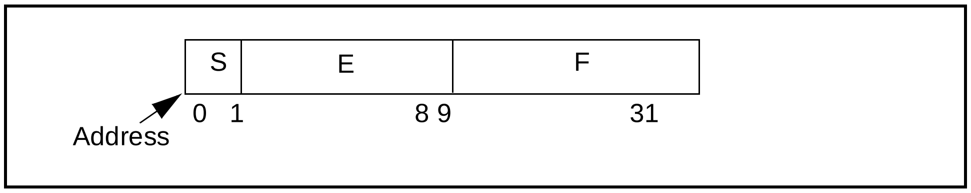

Figure B-8. Single-Precision Floating-Point Format

S is an unsigned single-bit sign field, E is an unsigned 8-bit exponent, and F is the unsigned fraction (mantissa).

If E = 0 and F = 0, the value is 0.

If E = 255 and F not = 0, the value is NaN. (*NaN* is IEEE's abbreviation for Not-a-Number.)

If E = 255 and F = 0, the value is $(-1)^{s} \infty$

If E = 0 and F not = 0, the value is $(-1)^{s}2^{-126}(0.F)$
[denormalized].

Otherwise, the value is $(-1)^{s}2^{E-127}(1.F)$.

The 64-bit double-precision floating-point format is shown in Figure B-9.

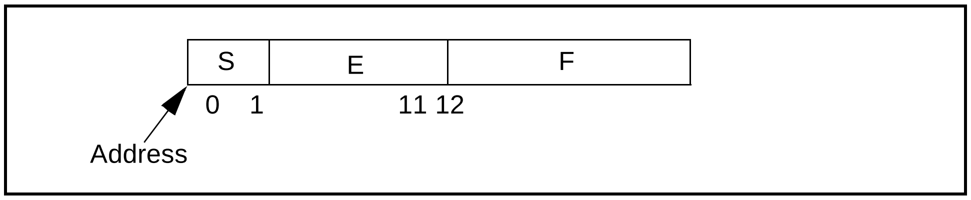

Figure B-9. Double-Precision Floating-Point Format

S is an unsigned single-bit sign field, E is an unsigned 11-bit exponent, and F is the unsigned fraction (mantissa).

If E = 0 and S is 0, the value of the number equals 0.

If E = 2047 and F not = 0, the value is NaN.

If E = 2047 and F = 0, the value is $(-1)^{s} \infty$

If E is 0 and F not = 0, the value is $(-1)^{s}2^{-1022}(0.F)$
[denormalized].

Otherwise, the value is $(-1)^{s}2^{E-1023}(1.F)$.

The following C/C++ data types take up a 32-bit single-precision floating-point word: `float`. The following C/C++ data types take up a 64-bit double-precision floating-point word: `double`, `long double`.

C/C++ Data Types

For a summary of the sizes and value ranges of the C/C++ data types, see Table A-1 and Table A-2

Index

Spine for 1” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up

PowerMAX OS

Progr

C/C++ Ref Manual

0890497