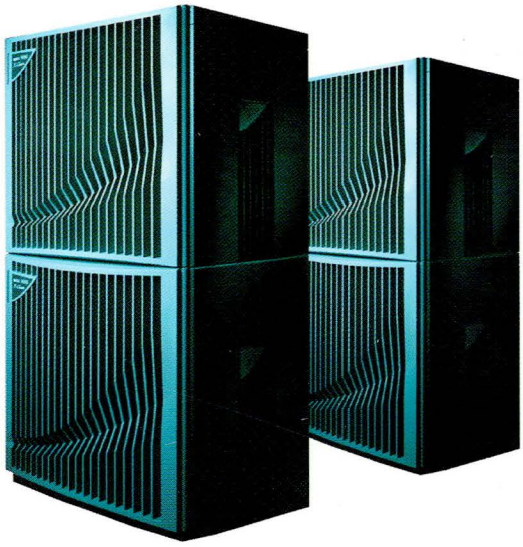
HEWLETT® PACKARD

S-Class and
X-Class Servers



Exemplar C++ Programming Guide

Third Edition

Hewlett-Packard Company
Convex Division
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America

Exemplar C++ Programming Guide

S-Class and X-Class Servers

B5630-90001

Third Edition

January 1997

Hewlett-Packard Company
Convex Division
Richardson, Texas
United States of America

Exemplar C++ Programming Guide

S-Class and X-Class Servers

B5630-90001

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

This entire book is recyclable.

Printed in the United States of America

Revision Information for

Exemplar C++ Programming Guide

S-Class and X-Class Servers

Edition	Document No.	Description
Third	B5630-90001	Updated for V10.23 of the Exemplar C++ compiler, released January, 1997.
Second	720-008130-001	Updated for V3.75 of the C++ compiler, released June, 1996.
First	720-008130-000	Initial release September, 1995.

Contents

Figures

Tables

How to use this guide

Purpose and audience

This guide describes how to use the Exemplar C++ compiler on Exemplar S-Class and X-Class servers. It describes the differences between the Exemplar C++ compiler running on SPP-UX and the standard Hewlett-Packard (HP) C++ compiler on which it is based. All Exemplar-specific features are described in detail. Examples showing how to develop C++ applications for Exemplar systems are included throughout the book.

The target audience for this book is the experienced C++ programmer who has a basic familiarity with SPP-UX, HP-UX, or UNIX.

Organization

This guide is organized as follows:

- Chapter 1, *Introduction*, is an overview of the Exemplar C++ product architecture and describes the preprocessor, translator, optimizer, and code generator components of the product.
- Chapter 2, *Compiling and executing C++ programs*, describes how to compile and execute C++ programs and describes the Exemplar C++ compiler options, optimization features, and system libraries.
- Chapter 3, *Using software development tools*, describes the development tools — debuggers and profilers — available from Hewlett-Packard for Exemplar C++ program development. This chapter gives brief descriptions of the tools and provides references to the documentation for each tool.
- Chapter 4, *Inter-language communication*, describes guidelines for linking Exemplar C++ modules with modules written in Exemplar C, standard HP C, Exemplar Fortran 77, and standard HP Fortran 77.

- Chapter 5, *Developing parallel C++ applications*, describes the programming paradigms used in developing PVM, MPI, and GSM parallel C++ applications.

Notational conventions

This section discusses notational conventions used in this book.

`Bold monospace`

In command examples, text shown in **`bold monospace`** identifies user input that must be typed exactly as shown.

`Monospace`

In paragraph text, `monospace` identifies command names, system calls, and data structures and types.

In command examples, `monospace` identifies command output, including error messages.

In command syntax diagrams, text shown in `monospace` must be typed exactly as shown.

Italic

In paragraph text, *italic* identifies new and important terms and titles of documents.

In command syntax diagrams, *italic* identifies variables that must be supplied by the user.

{ }

In command syntax diagrams, text surrounded by curly brackets indicates a choice. The choices available are shown inside the curly brackets and separated by the pipe (|) sign.

The following command example indicates that you can enter either `a` or `b`:

```
command {a | b}
```

[ ]

In command syntax diagrams, square brackets indicate optional data.

The following command example indicates that the variable *output_file* is optional:

`command` *input_file* [*output_file*]

...

In command syntax, horizontal ellipsis shows repetition of the preceding item(s).

The following command example indicates you can optionally specify more than one *input_file* on the command line:

`command` *input_file* [*input_file ...*]

KEYCAP

In paragraph text, text shown in **KEYCAP** indicates keyboard keys you must press to execute the command. For example, **RETURN** refers to the carriage return key.

Two **KEYCAP** terms separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-d** indicates that you must press the **d** key while holding down the **CTRL** key.

Notes

This document presents notes in the following format:

Note

A Note highlights supplemental information.

Associated documents

Using this software may require information not specific to the tasks described in this document.

For more information about the C++ programming language, the UNIX System Laboratories (USL) and HP compilers on which the Exemplar C++ compiler is based, and the software tools that you can use to develop C++ applications, see the following documents:

- *ADB Tutorial* (92432-90005). This book describes the Hewlett-Packard `adb` assembly-language debugger.
- *CXdb Quick Reference* (B5639-90001). This book describes the Exemplar CXdb visual debugger.
- *CXpa Reference* (B5639-90002). This book describes the Exemplar CXpa performance analyzer.
- *The Annotated C++ Reference Manual.* Ellis, Margaret A. and Bjarne Stroustrup. Murray Hill, NJ: Addison-Wesley. 1990. This is a standard reference manual for the C++ programming language.
- *Exemplar Programming Guide* (B5600-90001). This book describes efficient programming techniques for Exemplar systems.
- *HP C++ Quick Reference Card.* This quick reference card provides a brief overview of the C++ programming language.
- *HP C++ Programmer's Guide* (92501-90005). This book describes how to compile, run, and debug HP C++ programs.
- *HP Codelibs Library User manual.* This book describes how to use the HP Codelibs library, a general-purpose library package for use with C++.

- *HP-UX Symbolic Debugger User's Manual.* This manual describes how to debug programs on the HP-UX operating system using the `xdb` symbolic debugger.
- *HP/DDE Debugger User's Manual.* This manual describes how to debug programs on the HP-UX operating system using the `dde` symbolic debugger.
- *Programming on HP-UX* (B2355-90652). This book describes how to develop software on HP-UX using the HP compilers, assemblers, linker, libraries, and object files.
- *HP PVM User's Guide* (B5885-90001). This book describes the Hewlett-Packard implementation of the PVM system and explains how to write message-passing programs for Exemplar systems.
- *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing* (B5885-90002). Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. Cambridge, MA: MIT Press, 1994. This book provides an overview and tutorial of the PVM message-passing parallel programming system.
- *HP MPI User's Guide* (B6011-90001). This book describes the Hewlett-Packard implementation of the MPI standard and explains how to write message-passing programs for Exemplar systems.
- *Using MPI — Portable Parallel Programming with the Message-Passing Interface* (B6011-90003). Gropp, William, Ewing Lusk, and Anthony Skjellum. Cambridge, MA: MIT Press, 1994. This book provides an overview and tutorial of the MPI message-passing parallel programming standard.
- *The C++ Programming Language,* 2nd ed. Stroustrup, Bjarne. Murray Hill, NJ: Addison-Wesley. 1992. This book provides an overview of the C++ programming language.
- *USL C++ Language System Library Manual.* This book describes the C++ class libraries provided with the USL C++ compiler; the Exemplar C++ compiler provides the same class libraries.
- *USL C++ Language System Product Reference Manual.* This book describes the C++ language implementation of the USL C++ compiler; the Exemplar C++ compiler uses the same implementation.

Ordering documents

To order the current edition of these or any other Exemplar documents, send requests to:

Hewlett-Packard Company
Convex Division
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Please include the order number or the exact title of the document.

Technical assistance

If you have questions that are not answered in this book, contact the Hewlett-Packard Convex Technical Assistance Center (TAC) at the following locations:

Within the continental U.S., call 1 (800) 952-0379.

From Canada, call 1 (800) 345-2384.

All other locations, contact the local Hewlett-Packard office.

You can also use the `contact` utility if you would like to report any problems you may have with the Exemplar C++ product or its documentation. For more information, refer to the contact(1) man page.

Introduction

1

The Exemplar C++ compiler is a complete implementation of the C++ programming language described in *The Annotated C++ Reference Manual* by Ellis and Stroustrup. This book documents version 10.23.02 of the Exemplar C++ compiler, which runs on Exemplar S-Class and X-Class servers.

The Exemplar C++ compiler translates a source file containing one or more C++ program units into an object module. The object module then can be linked with library routines or other object modules for execution on SPP-UX V5.0 or higher running on an SPP1200, SPP1600, S2000 server, or X2000 server. Previously compiled programs written in assembly language, C, Fortran, or C++ can also be linked with Exemplar C++ object code to produce an executable program.

The Exemplar C++ compiler uses the same optimizer and code generator as the Exemplar C and Fortran compilers. All three of these compilers support the Exemplar programming model, which features multithreaded parallelism and global shared memory (GSM). The Exemplar compilers automatically generate code that takes advantage of the architecture of the Exemplar S-Class and X-Class servers.

The Exemplar C++ compiler operates in two different modes: *compiler mode* and *translator mode*. The two modes are described in the following sections. Compiler mode is the default mode.

Compiler mode

In compiler mode, the Exemplar C++ compiler converts C++ source code directly to object code. The resulting executable file executes only on SPP-UX V5.0 or higher running on an SPP1200, SPP1600, S2000 server, or X2000 server. The steps of the compilation process for compiler mode are shown in Figure 1.

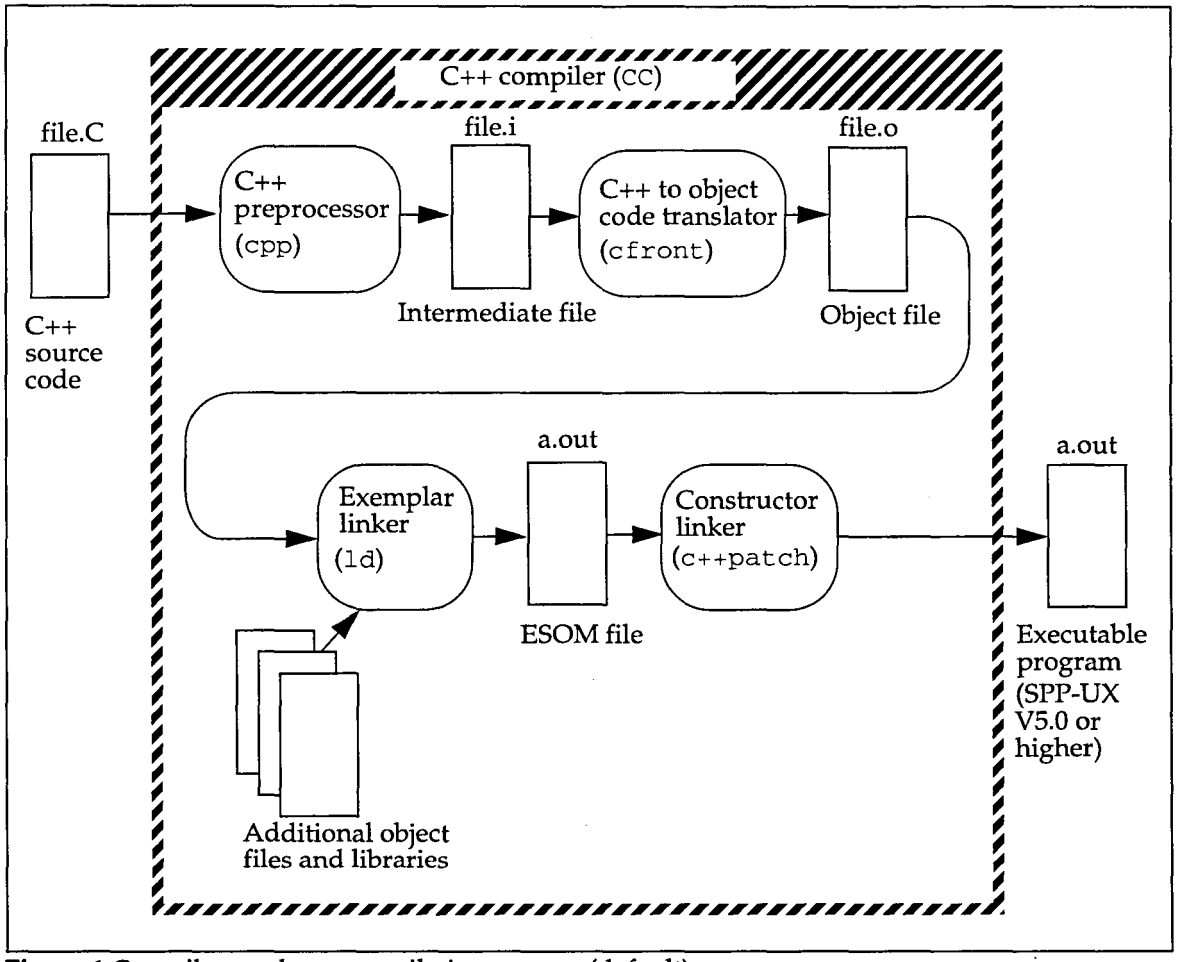


Figure 1 Compiler mode CC compilation process (default)

Translator mode

In translator mode, the Exemplar C++ compiler translates C++ source code into C source code, then compiles the C source code using the Exemplar C compiler. To use the compiler in translator mode, specify the `+T` option on the C++ compiler (`CC`) command line. The resulting executable file executes only on SPP-UX V5.0 or higher running on an SPP1200, SPP1600, S2000 server, or X2000 server. The steps of the compilation process for translator mode are shown in Figure 2.

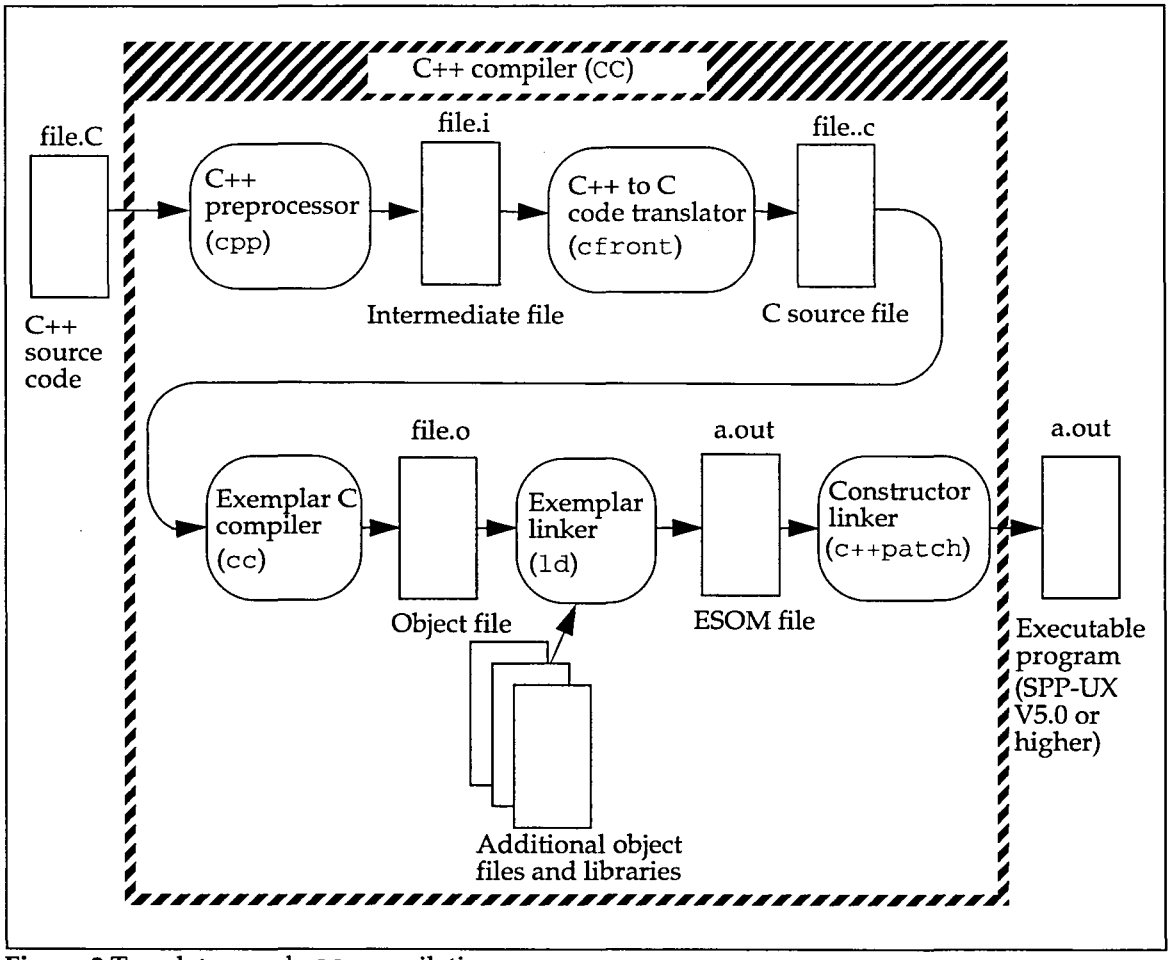


Figure 2 Translator mode `CC` compilation process

C++ compilation steps

The following sections describe the steps involved in compiling a C++ source file.

Preprocessor

The Exemplar C++ compiler first invokes the `cpp` preprocessor on all source files that have a file name suffix that begins with .c or .C. The preprocessor examines all lines beginning with a # character, performs the corresponding actions and macro replacements, and produces a preprocessed version of your program with the file name suffix .i. The .i file is created in a directory used to store temporary files. If the next phase of the compile process is successful, the .i file in the temporary directory is deleted by default. Use the `-P` compiler option to save the .i files. (If you use the `-P` option, no other phases of compilation are performed.)

C++ front-end processor

The Exemplar C++ compiler passes the intermediate (.i) file created by the preprocessor to the `cfront` front-end processor. The action `cfront` performs depends on whether you are using the compiler in translator mode or compiler mode.

`cfront` operation in compiler mode

In compiler mode, `cfront` compiles the intermediate (.i) file and generates object code in a file with the file name suffix .o. (See Figure 1.)

`cfront` operation in translator mode

In translator mode, `cfront` translates the intermediate (.i) file into C source code, and the resulting intermediate C source file is passed to the Exemplar C compiler. (See Figure 2.)

C compiler (translator mode only)

In translator mode, the intermediate C source file created by the `cfront` processor is compiled by the Exemplar C compiler `cc`, using the applicable compiler options specified in the `cc` (C++ compiler) command. The C compiler generates object code in a file with the file name suffix .o.

Linker

In both translator and compiler mode, the .o file is passed to the `ld` linker program. The linker resolves external references, searches libraries to resolve references to library routines, and combines object files into an executable program file.

The output of the Exemplar `ld` program is an a.out file in extended shared object module (ESOM) format, which supports multithreaded parallel execution as well as the Exemplar debugging and profiling tools.

Constructor linker

In both translator and compiler mode, the a.out file is passed to the `c++patch` constructor linker. The constructor linker chains constructors and destructors of static objects in the executable file.

The Exemplar `c++patch` program produces an executable file that is in ESOM format. ESOM executables execute on SPP-UX V5.0 or higher running on an SPP1200, SPP1600, S2000 server, or X2000 server.

Compiling and executing C++ programs

2

This chapter describes the options that you can specify on the command line of the Exemplar C++ compiler and explains how they are used.

Overview

The current release of the Exemplar C++ compiler conforms to the definition of the C++ language provided in *The Annotated C++ Reference Manual* by Ellis and Stroustrup. Programs written for Exemplar C++ can be compiled by other C++ compilers conforming to this definition with little or no modification to the source code; the converse is also true.

The Exemplar C++ compiler can generate code that takes advantage of the architecture of the Exemplar family of technical servers. The Exemplar C++ compiler has options that allow code to be optimized for a specific target system — SPP-UX V5.0 or higher running on an SPP1200, SPP1600, S2000 server, or X2000 server.

Compiling programs

The CC command invokes the Exemplar C++ compiler. The CC command has the following format:

CC [*options*] *files*

options

is an optional argument specifying one or more of the options described in the remaining sections of this chapter.

files

represents one or more source files to be compiled, object files to be linked, assembly-language files to be assembled, or libraries to be linked.

File-naming conventions

The compiler identifies certain arguments as file names based on the file extension.

A C++ source file is identified by the file extension that begins with .c or .C. Each compiled object file has the same name as the source file except that it has a .o file extension. However, if a single C++ source file is compiled and linked in one step, the .o file is deleted.

Arguments that end with .s are understood to be assembly-language source files. These files are assembled, producing a .o file for each .s file.

Arguments that end in .i are identified as output files from the C++ preprocessor, cpp. These files are compiled again without first invoking cpp, producing a .o file for each .i file. For more information, see the -P option in this chapter and the cpp(1) man page.

Arguments of the form -l*x* cause the linker to search the library lib*x*.sl or lib*x*.a in an attempt to resolve currently unresolved external references. Because a library is searched when its name is encountered, placement of a -l argument on the command line is significant. If a file contains an unresolved external reference, the library containing the definition must be placed after the file on the command line. See the ld(1) man page for more information.

All other arguments, such as those whose names end with .a, .o, or .sl, are assumed to be relocatable object files or libraries and are passed to ld to be included in the link operation.

Table 1 summarizes the file-naming conventions used by Exemplar C++.

Table 1 C++ file name extensions

File type	File extension
C++ source files	.c or .C
Assembly-language source files	.s
Files created by the cpp preprocessor	.i
Library files	.a or .sl
Relocatable object files	.o

Using environment variables

The following environment variables are used to provide information to the C++ compiler.

CXXOPTS environment variable

Arguments and options can be passed to the compiler through the CXXOPTS environment variable as well as on the command line. The compiler reads the value of CXXOPTS and divides the arguments into two sets: those that appear before the vertical bar (|) and those that appear after it. The first set of arguments is placed before any command-line parameters to CC, and the second set of arguments is placed after any command-line parameters to CC. If you do not use the vertical bar, all arguments are placed before the command-line parameters.

For example, the following commands (using C shell notation)

```
setenv CXXOPTS -v | -lmalloc
CC -g prog.C
```

are equivalent to the following:

```
CC -v -g prog.C -lmalloc
```

LDPATH environment variable

If LDPATH is set before CC is invoked, ld will use the value of LDPATH, along with any directories specified in the -L command-line option, as the search path for libraries. Otherwise, CC automatically constructs an appropriate library search path based on the command-line options.

TMPDIR environment variable

The TMPDIR environment variable can be set to specify a directory to be used for temporary files. This directory overrides the default temporary directory /var/tmp.

Compiler options

CC recognizes the following compiler options. Unrecognized options cause a warning message to be printed to standard error.

+a{0 | 1}

Specifies which style of declarations to produce. The compiler can produce either ANSI C style or "classic C" (also known as K&R C) style declarations. The +a0 option causes the compiler to produce "classic C" style declarations. The +a1 option causes the compiler to produce ANSI C style declarations and follow ANSI rules in regard to promotion of floats to doubles.

The default is +a0 unless the -cxdb option is specified. When -cxdb is specified, the default is +a1. Do not use the +a0 option with the -cxdb option; this will hinder the debugger's ability to resolve overloaded functions.

+A

Causes the linker ld to use only archive libraries for all occurrences of the -l option. The +A option informs the C++ runtime environment that no shared libraries will be used with the program. Do not use the +A option when calling shl_load(3) either directly or indirectly.

-A*level*

Selects the mode of preprocessor operation. *level* can have one of the following values:

a

Requests the ANSI mode preprocessor (the default mode).

c

Requests the compatibility (K&R) mode preprocessor.

-b

Causes the linker ld to create a shared library rather than a normal executable file. Object files processed with this option must contain position independent code (PIC). Either the +z or +Z option must be used to produce position independent code. In the default mode, CC does not generate position independent code. See the ld(1) man page for more information.

`-c`

Suppresses the link edit phase of the compilation and forces an object (.o) file to be produced for each .c or .C file even if only one program is compiled. Object files produced from C++ programs must be linked before being executed.

`-C`

Prevents the preprocessor from stripping comments. See the cpp(1) man page for more information.

`+d`

Prevents the expansion of inline functions.

`+df`*name*

Specifies profile database file *name* for profile-based optimizations. The default is flow.data if *name* is not specified. No white space is permitted between `+df` and *name*. Data for more than one application can be kept in the same file. `+df` requires the specification of either `+I` or `+P`. See the descriptions of the `+I`, `+P`, and `+pgm` options and the ld(1) man page for more information.

`-D`*name=def* or `-D`*name*

Defines *name* to the preprocessor. This has the same effect as a `#define` statement. See the cpp(1) man page for more information.

`+e{0 | 1}`

Optimizes a program to use less space by ensuring that only one virtual table is generated per class. The `+e0` option causes virtual tables to be external and defined elsewhere (that is, uninitialized). The `+e1` option causes virtual tables to be declared externally and defined in the current module (that is, initialized). When neither option is used, virtual tables are static and there is one per file. Usually, `+e1` is used to compile one file that includes class definitions, while `+e0` is used on all the other files including those class definitions.

`+eh`

Enables exception handling. Exception handling is supported in both compiler mode and translator mode, and object files created in the two modes can be mixed. However, code that has been compiled with the `+eh` option is not link compatible with code that has not been compiled with the `+eh` option. Attempts to link these two kinds of object files together will generate link (or runtime) warnings. Use the `+eh` option when linking objects compiled with the `+eh` option. With the `+eh` option, the `CC` driver will automatically link to the `+eh` version of the runtime library.

`-ext`

Enables extensions to Exemplar C++ to support the data type `long long`, which is for 64-bit integers.

`-E`

Runs only `cpp` on the named C++ or assembly-language source files, and sends the result to the standard output.

`-F`

Runs only `cpp` and `cfront` on the named C++ source files, and sends the result to the standard output. The `-F` option implies the `+T` option.

`-Fc`

This option operates like the `-F` option, but the output is C source code suitable as a .c file for `cc`. This option is equivalent to the `-F` and `+L` options. The `-Fc` option implies the `+T` option.

-.suffix

Instead of using the standard output for the `-E`, `-F`, or `-Fc` options, this option places the output from each .c file into a file with the specified *suffix*.

`-g`

Causes the compiler to generate additional information needed by the Exemplar symbolic debugger, CXdb. This option forces compilation at optimization level `+O0` regardless of any other compiler options.

`-g1`

Causes the compiler to generate less symbolic debug information than with the `-g` option, thereby decreasing the size of the object file. This option should only be used to compile an entire application.

Specifically, the `-g` option emits full debug information for every class referenced in a file, which can result in redundant information. The `-g1` option causes complete class information to be generated only for the file in which the class is defined (the first non-inline, non-pure virtual function specifies the definition). If an entire application is compiled with the `-g1` option, no debugger functionality is lost.

`+i`

Causes an intermediate ..c (two periods followed by a 'c') C language source file to be created in the current directory. The `+i` option must be used with the `+T` option.

`+I`

Instruments the application for profile-based optimization. See the descriptions of the `+df`, `+P`, and `+pgm` options and the ld(1) man page for more information. This option is incompatible with the `-g`, `-G`, `-gl`, `-s`, `-S`, `-y`, and `+eh` options.

`-I`*dir*

Changes the algorithm used by the preprocessor for finding include files, causing it to search also in directory *dir*. See the cpp(1) man page for more information.

`+k`

Generates the proper instruction sequence for accessing shared global data items when the program uses a large number of global data items. The linker issues an error message when this option is needed.

`-l`*x*

Causes the linker to search the library lib*x*.sl or lib*x*.a in an attempt to resolve currently unresolved external references. Because a library is searched when its name is encountered, placement of a `-l` option is significant. If a file contains an unresolved external reference, the library containing the definition must be placed after the file on the command line. See the ld(1) man page for more information.

`+L`

Generates source line number information using the format `#line %d` instead of `#%d`.

`-L`*dir*

Changes the algorithm used by the linker to search for libraries. The `-L` option causes `ld` to search in *dir* before searching in the default locations. This option is effective only if it precedes the `-l` option on the command line. See the ld(1) man page for more information.

`+m`

Provides maximum compatibility with the USL C++ implementation. Exemplar C++ provides optimizations and additional functionality that may not be compatible with other C++ implementations.

`-n`

Causes the output file from the linker to be marked as shareable. See the ld(1) man page for more information and system defaults.

`-N`

Causes the output file from the linker to be marked as unshareable. See the ld(1) man page for more information and system defaults.

`-o`*outfile*

Sets the name of the linker output file to *outfile*. The default file name is a.out.

`-O`

Invokes the optimizer with level 2 optimization. This option is equivalent to `+O2`. See "Optimization options" on page 21 for more information about optimization options.

`+p`

Disallows all anachronistic constructs. Ordinarily the compiler gives warnings about anachronistic constructs; using the `+p` option, the compiler will not compile code containing anachronistic constructs. See the *USL C++ Reference Manual* for a list of anachronisms.

`+pa`

Instruments the compiled code for profiling with the Exemplar performance analyzer, CXpa. This option is not compatible with the `-p` and `-G` options. For more information about the optional product CXpa, refer to the *CXpa Reference* or the cxpa(1) man page.

`+pgm`*name*

Specifies a profile database lookup name within the database file name. No white space is permitted between `+pgm` and *name*. `+pgm` requires that either `+I` or `+P` be specified. See the descriptions of the `+df`, `+I`, and `+P` options and the ld(1) man page for more information.

`-pta`

Instantiates an entire template class rather than only those members that are used. This option is incompatible with the `-pts` option.

`-ptb`

Uses ld(1) instead of nm(1) to do the link simulation step. This option might be necessary for linking with dependent shared libraries.

`-pth`

Specifies that template instantiation files should be created using short file names. (Template instantiation files are object files created in the template repository by `c++ptlink`.) Exemplar C++ creates template instantiation files using long

file names by default. See the convertfs(1M) man page for more information about long file names.

`-ptH"`*list*`"`

Specifies a list of file name extensions that template declaration files (header files) can have. When compiling or instantiating templates, the compiler searches for header files with these extensions in the order the extensions are listed. For example, `-ptH".h .H"` specifies that template declaration header files can have extensions of .h or .H. By default, Exemplar C++ uses the following list of extensions: `".h .H .hxx .HXX .hh .HH .hpp"`.

`-ptn`

Changes the default instantiation behavior for one-file programs to that of larger programs, where instantiation is broken out separately and the repository is updated. One-file programs normally have instantiation optimized so that templates are instantiated directly into the application object file itself.

`-ptr`*pathname*

Specifies a repository for templates. The default repository is ./ptrepository. If several repositories are given, they will be searched in the order in which they are listed. At link time, repositories are ignored if they do not have a valid `defmap` file. Only the first repository is writable, and all others are readable.

`-ptR`

Specifies that `c++ptcomp` always use <> instead of `""` as delimiters for include files in `defmap`. This option also specifies that `c++ptlink` ignore any differences in the `-D` or `-I` options. The `-ptR` option can prevent re-instantiation of the same template(s) when compiling and linking from multiple directories and using the same repository. It can also prevent "out-of-date" errors on secondary repositories. You might have to add the `-I` option to refer to the current directory. The `-ptR` option assumes that there are no differences in the preprocessed source code even if the `-D` or `-I` options are changed.

`-pts`

Splits instantiations into separate object files, with one function per object (including overloaded functions) and with all static data and virtual functions grouped into a single object. This option is incompatible with the `-pta` option. The `-pts` option can be used to split up only needed functions rather than all functions.

`-ptS"`*list*`"`

Specifies a list of file name extensions that template definition files (source files) can have. When compiling or instantiating templates, the compiler searches the source files with these extensions in the order in which the extensions are listed. For example, `-ptS".c .C"` specifies that template definition files can have extensions of .c or .C. By default, Exemplar C++ uses the following list of extensions: `".c .C .cxx .CXX .cc .CC .cpp"`.

`-ptv`

Turns on verbose or verify mode, which displays each phase of instantiation as it occurs. Verbose mode displays the reason for instantiation and the exact `CC` command used.

`+P`

Optimizes the application based on profile data found in the database file flow.data, produced by executing a program compiled with `+I`. See the descriptions of the `+df`, `+I`, and `+pgm` options and the ld(1) man page for more information. This option is incompatible with `-G`, `-g`, `-g1`, `-S`, `+I`, `-y`, and `+eh`.

`-P`

Runs only `cpp` on the named C++ files and places the result in corresponding files with a .i suffix.

`-q`

Causes the output file from the linker to be marked as demand loadable. See the ld(1) man page for more information and system defaults.

`-Q`

Causes the output file from the linker to be marked as not demand loadable. See the ld(1) man page for more information and system defaults.

`-s`

Causes the output of the linker to be stripped of symbol table information. See the strip(1) man page for more details. The use of this option prevents the use of a symbolic debugger on the resulting program. See the ld(1) man page for more information.

`-S`

Compiles the named C++ files and leaves the assembly language output in corresponding files with a .s suffix.

`-tm` *target*

Specifies the target machine architecture for the executable code. *target* can have one of the following values:

`spp1200`

Executable code runs on SPP-UX V5.0 or higher running on an SPP1200, SPP1600, S2000 server, or X2000 server. Instruction scheduling is optimized based on the SPP 1200 instruction timings. The SPP versions of libraries are selected at link time. If you compile with the `+O3 +Oparallel` options, the `CPSlib` library is included in the linkage sequence, allowing threaded applications to be created. Executables and libraries produced with this option are in ESOM format; they will not run under HP-UX.

`spp1600`

Executable code runs on SPP-UX V5.0 or higher running on an SPP1200, SPP1600, S2000 server, or X2000 server. Instruction scheduling is optimized based on the SPP 1600 instruction timings. The SPP versions of libraries are selected at link time. If you compile with the `+O3 +Oparallel` options, the `CPSlib` library is included in the linkage sequence, allowing threaded applications to be created. Executables and libraries produced with this option are in ESOM format; they will not run under HP-UX.

`S2000`

Executable code runs on any Exemplar S2000 or X2000 server. Instruction scheduling is optimized based on the instruction timings of the S-Class server. The Exemplar versions of libraries are selected at link time. If you compile with the `+O3 +Oparallel` options, the `CPSlib` library is included in the linkage sequence, allowing threaded applications to be created. Executables and libraries produced with this option are in ESOM format; they will not run under HP-UX.

`X2000`

Executable code runs on any Exemplar S2000 or X2000 server. Instruction scheduling is optimized based on the instruction timings of the X-Class server. The Exemplar versions of libraries are selected at link time. If you compile with the `+O3 +Oparallel` options, the `CPSlib` library is included in the linkage sequence, allowing threaded applications to be created. Executables and

libraries produced with this option are in ESOM format; they will not run under HP-UX.

The default value of *target* is the type of machine on which the compiler is currently running. The `+DS1.1` option forces the target architecture to spp1600, and the `+DS2.0` option forces the target architecture to X2000, regardless of any other compiler option settings.

`-t`*x*,*name*

Substitutes or inserts subprocess *x* with *name*, where *x* is one or more of a set of identifiers indicating the subprocess or subprocesses. This option works in two modes: if *x* is a single identifier, *name* represents the full path name of the new subprocess; if *x* is a set of identifiers, *name* represents a prefix to which the standard suffixes are concatenated to construct the full path names of the new subprocesses. *x* can have one or more of the following values:

Table 2 `-t` parameter values

value	description	suffix
`p`	Preprocessor	cpp or cpp.ansi
`C`	Compiler/translator body	cfront
`r`	Compile-time template processor	c++ptcomp
`i`	Link-time template processor	c++ptlink
`c` or `0`	Compiler body in translator mode	ccom
`a`	Assembler	as
`l`	Linker	ld
`b`	Code generator in translator mode	cc
`m`	Merge tool	c++merge
`f`	Filter tool	c++filt
`P`	Patch tool	c++patch
`u`	Standalone code generator	uCcom
`x`	All subprocesses	

`+T`

Requests translator mode. In translator mode, C++ source code is translated to C code which is then compiled by `cc` to object code.

`-U`*name*

Removes any initial definition of *name* in the preprocessor. See the cpp(1) man page for more information.

`-v`

Enables verbose mode, which outputs a step-by-step description of the compilation process to standard error.

`+w`

Warns about all questionable constructs. Without the `+w` option, the compiler issues warnings only about constructs that are almost certainly problems.

`-w`

Suppresses warning messages.

`-W`*x*,*arg1*[,*arg2*...]

Hands off the argument(s) *arg1*[,*arg2*...] to pass *x*, where *x* can assume one of the values listed under the `-t` option except `u` (standalone code generator). *x* can also have the value `d` (driver program). The `-W` option specification allows additional, implementation-specific options to be recognized by the compiler driver.

`+x`*file*

Reads a file of sizes and alignments. Each line contains three fields: a type name, the size (in bytes), and the alignment (in bytes). This option is useful in cross compilations.

`-y`

Generates additional information needed by static analysis tools and ensures that the program is linked as required for static analysis. This option is incompatible with the optimization options and the `+T` option.

`-Y`

Enables support of 8-bit, 16-bit, and 4-byte EUC characters in comments, string literals, and character constants.

`+z, +Z`

Causes the compiler to generate position independent code (PIC) for use in building shared libraries. The options `-G`, `-p`, and `+pa` are ignored if used with PIC. When certain limits are exceeded, `+Z` is required to generate PIC. The linker `ld` issues the error indicating when `+Z` is required. If both `+z` and `+Z` are specified, only the last one encountered applies. For a more complete discussion regarding PIC and these options, see the manual *Programming on HP-UX*.

`-z`

Does not bind anything to address zero. This option allows runtime detection of `NULL` pointers.

`-Z`

Allow dereferencing of `NULL` pointers.

Referencing `NULL` pointers

Accessing the object of a `NULL` (zero) pointer is technically illegal, but many C++ systems have permitted it in the past. The `-z` and `-Z` options are provided to allow portability of this code. If the hardware is able to return zero for reads of location zero, at least when referencing 8-bit and 16-bit quantities, it must do so unless the `-z` flag is present. The `-z` flag requests that SIGSEGV be generated if an access to location zero is attempted. Writes of location zero may be detected as errors, even if reads are not. If the hardware cannot assure that location zero acts as if it was initialized to zero or is locked at zero, the hardware acts as if the `-z` option is always set.

Instantiating templates

The template instantiating system has dependencies on options passed to `CC` on the command line. Options that are normally specified to compile an application, such as `-D` and `-I`, must also be specified at link time so that the template instantiation system can instantiate template types with the appropriate header files and options.

In the following example, the `-I` option used in the compile step is repeated in the link step:

```
CC -c -I./includefiles myprog.C
CC -I./includefiles myprog.o
```

Optimization options

The compiler options listed in this section allow the generation of code that is optimized for a particular system architecture.

The `-g` and `-gl` options are incompatible with optimization. If both debug and optimization options are specified, the debug option takes precedence and optimization does not take place.

`-depth`

Instructs the runtime system to traverse the shared library list in a depth-first manner instead of the default left-to-right search when calling static constructors. The traversal of the static constructor chain within each shared library is not affected by this option.

`+DA`*architecture*

Generates code for the *architecture* specified. The default code generated for the Series 800 is PA-RISC 1.0. The default code generated for the Series 700 is PA-RISC 1.1. The default code generated for Exemplar S-Class and X-Class servers is PA-RISC 2.0. The default code generation can be overridden using the `CXXOPTS` environment variable or the `+DA` compiler option. *architecture* can be either a model number (for example, 750 for the HP 9000/750 or 870 for the HP 9000/870) or one of the following generic specifications:

`1.0`

Precision architecture RISC, version 1.0 or higher. This is the default for all Series 800 models.

`1.1`

Precision architecture RISC, version 1.1. This is the default for all Series 700 models.

`2.0`

Precision architecture RISC, version 2.0. This is the default for Exemplar S-Class and X-Class servers.

The compiler determines the target architecture using the following precedence:

1. Specification of the `+DA` option on the compiler command line.
2. Specification of `+DA` in the `CXXOPTS` environment variable.
3. Specification of the `-tm` option, which selects a value for `+DA` corresponding to the processor of the target machine.
4. The default as mentioned above.

+DS*architecture*

Uses the instruction scheduler tuned to the *architecture* specified. If this option is not specified, the compiler uses the instruction scheduler for the architecture on which the program is compiled. The architecture is determined by `uname()`; see the uname(2) man page for more information. *architecture* can be a model number (for example, 750 for the HP 9000/750 or 870 for the HP 9000/870). See /usr/lib/sched.models for a list of HP model numbers and processor names.

+O*opt*

Invokes the optimizations identified by optimization level *opt*. The values of *opt* have the following meanings:

Table 3 Optimization options

opt	Description
0	Perform minimal optimizations. This is the default.
1	Perform optimizations within basic blocks only.
2	Perform level 1 and global optimizations. This is the same as -O.
3	Perform level 2 as well as interprocedural global optimizations.
4	Perform level 3 as well as link-time optimizations.

+R*num*

Allow only the first *num* `register` variables to actually have the `register` class. Use this option when the register allocator issues an `out of general registers` message.

The following optimization options allow you to enable or disable specific optimization techniques.

`+O[no]autopar`

When used with `+Oparallel` option, `+Oautopar` causes the compiler to automatically parallelize loops that are safe to parallelize. A loop is safe to parallelize if it has an iteration count determinable at runtime before loop invocation, and contains no loop-carried dependences, procedure calls, or I/O operations. A loop-carried dependence exists when one iteration of a loop assigns a value to an address that is referenced or assigned on another iteration.

The default is `+Onoautopar`. However, because parallelization takes places only at optimization level 3 and above, `+O[no]autopar` has effect only at levels 3 and 4.

`+O[no]dataprefetch`

When `+Odataprefetch` is enabled, the optimizer will insert instructions within innermost loops to explicitly prefetch data from memory into the data cache. For cache lines containing data that will be written, `+Odataprefetch` prefetches the cache lines so that they are valid for both read and write access. Data prefetch instructions will be inserted only for data referenced within innermost loops using simple loop varying addresses (that is, in a simple arithmetic progression). `+Odataprefetch` is effective only for PA-RISC 2.0 targets.

The math library libm contains special prefetching versions of vector routines. If you have a PA-RISC 2.0 application that contains operations on arrays larger than 1 megabyte in size, using `+Ovectorize` in conjunction with `+Odataprefetch` may improve performance substantially.

Use the `+Odataprefetch` option for applications that have high data cache miss overhead. This option is available at optimization levels 2, 3, and 4. The default is `+Onodataprefetch`.

`+O[no]dynsel`

When specified with `+Oparallel`, the `+Odynsel` option enables workload-based dynamic selection. This optimization causes the compiler to generate parallel and serial versions of parallelizable loops whose iteration counts are unknown at compile time. At runtime, the loop's workload is compared to parallelization overhead, and the parallel version is run only if it is profitable to do so.

The `+Onodynsel` option disables dynamic selection and tells the compiler that it is profitable to parallelize all parallelizable loops.

`+O[no]entrysched`

Perform [do not perform] instruction scheduling on a subprogram's entry and exit code sequences. This optimization can occur at optimization levels 1, 2, 3, and 4. The default is `+Onoentrysched`.

`+O[no]exemplar_model`

Enable [disable] support for the Exemplar programming model. The `+Oexemplar_model` option allows you to use the command-line options that are part of the Exemplar programming model. (Refer to the *Exemplar Programming Guide* for more information on the programming model.) This option implies the `+Okernel_threads` option.

The `+Onoexemplar_model` option turns off support for the Exemplar programming model. If you use this option, the compiler ignores all other options and optimizations specific to the Exemplar programming model.

`+O[no]fastaccess`

Enable [disable] fast access to global data items. This option can occur at optimization levels 0, 1, 2, 3, and 4. The default is `+Onofastaccess` at optimization levels 0, 1, 2 and 3, and `+Ofastaccess` at optimization level 4.

`+O[no]fltacc`

Enable [disable] optimizations that cause inaccurate floating-point results. This option can occur at optimization levels 2, 3, and 4. The default is `+Ofltacc`.

`+O[no]info`

Provide [do not provide] feedback information about the optimization process. This option is most useful at optimization levels 3 and 4. The default is `+Onoinfo`. See also `+Oreport`.

`+O[no]initcheck`

Enable [disable] initialization to zero of any local, scalar, non-static variable that is uninitialized with respect to at least one path leading to its use. This optimization can occur at optimization levels 2, 3, and 4. The default is to enable initialization if the variable is uninitialized with respect to every path leading to its use.

`+O[no]inline`

Enable [disable] optimizer inlining for any function. This option can occur at optimization levels 3 and 4. The default is `+Oinline`.

`+Okernel_threads`

Causes the compiler to use a thread-based model for parallelism. The Exemplar programming model requires thread-based parallelism. The default is `+Okernel_threads`. See also `+Oprocess_threads`.

`+O[no]libcalls`

Use [do not use] versions of selected library routines that provide low call overhead. No error checking is done; that is, errno(2) is not set. This option can be used at optimization levels 0, 1, 2, 3, and 4. The default is `+Onolibcalls`.

`+O[no]loop_transform`

Transform [do not transform] eligible loops for improved cache performance. This option can be used at optimization levels 3 and 4. The default is `+Oloop_transform`.

`+O[no]moveflops`

Enable [disable] moving conditional floating point instructions out of loops. This optimization can occur at optimization levels 2, 3, and 4. The default is `+Omoveflops`.

`+O[no]parallel`

Transform [do not transform] eligible loops for parallel execution on multiprocessor machines. Any files that are compiled with `+Oparallel` must also be linked with `+Oparallel`. This option can be used at optimization levels 3 and 4. The default is `+Onoparallel`.

`+O[no]parmsoverlap`

Optimize with the assumption that subprogram arguments do [not] refer to the same memory. This optimization can occur at optimization levels 2, 3, and 4. The default is `+Oparmsoverlap`.

`+O[no]pipeline`

Enable [disable] software pipelining. This optimization can occur at optimization levels 2, 3, and 4. The default is `+Opipeline`.

`+O[no]procelim`

Enable [disable] the elimination of procedures that are not referenced by the application. You can use this option at any optimization level. The default is `+Onoprocelim` at optimization levels 0, 1, 2, and 3. At optimization level 4, the default is `+Oprocelim`.

`+Oprocess_threads`

Causes the compiler to use process-based parallelism instead of thread-based parallelism. Process-based parallelism is the type used by the standard HP compilers. This option implies `+Onoexemplar_model`. If you specify both `+Oexemplar_model` and `+Oprocess_threads`, the compiler ignores `+Oprocess_threads` with a warning and selects `+Okernel_threads` instead.

`+O[no]regionsched`

Apply [do not apply] aggressive scheduling techniques to move instructions across branches. This optimization can occur at optimization levels 2, 3, and 4. The default is `+Onoregionsched`.

`+O[no]regreassoc`

Enable [disable] register reassociation. This option can occur at optimization levels 2, 3, and 4. The default is `+Oregreassoc`.

`+O[no]report[=`*report_type*`]`

Causes the compiler to display various types of optimization reports. This option can be used at optimization levels 3 and 4 only. The default is `+Onoreport`.

Table 4 lists the available report types. For more information about the reports, refer to the *Exemplar Programming Guide*.

Table 4 Optimization report types

report_type value	Report contents
`all`	Loop Report and Privatization Report
`loop`	Loop Report only (default)
`private`	Loop Report and Privatization Report

`+O[no]sharedgra`

Enable [disable] global register allocation for shared-memory variables that are visible to multiple threads. If you experience wrong answers due to a variable that is shared among parallel threads, use the `+Onosharedgra` option on any routine that references shared data and is potentially called from within a parallel region. This option is available at optimization levels 2, 3, and 4. The default is `+Osharedgra`.

`+O[no]signedpointers`

Enable [disable] the treating of pointers as signed quantities in comparisons. This option can occur at optimization levels 2, 3, and 4. The default is `+Onosignedpointers`.

`+O[no]volatile`

Enable [disable] the treating of all global variables as volatile quantities. This option can occur at optimization levels 1, 2, 3, and 4. The default is `+Onovolatile`.

The following prepackaged optimization options allow you to enable or disable groups of optimization options.

`+O[no]aggressive`

Apply [do not apply] aggressive optimizations; that is, new optimizations and the optimizations invoked by the following optimization settings:

`+Oentrysched`

`+Olibcalls`

`+Onofltacc`

`+Onoinitcheck`

`+Onoparmsoverlap`

`+Onoregionsched`

This optimization can occur at optimization levels 2, 3, and 4. The default is `+Onoaggressive`.

`+O[no]all`

Apply [do not apply] all optimizations enabled by the `+Oaggressive` and `+Onolimit` options. This option can occur at optimization level 4 only. The default is `+Onoall`.

`+O[no]conservative`

Make [do not make] conservative assumptions about the program when optimizing. This option can occur at optimization levels 2, 3, and 4. The default is `+Onoconservative`.

`+O[no]limit`

Suppress [do not suppress] optimizations that significantly increase compile time or consume large amounts of memory. This optimization can occur at optimization levels 2, 3, and 4. The default is `+Olimit`.

`+O[no]size`

Suppress [do not suppress] optimizations that significantly increase code size. This optimization can occur at optimization levels 2, 3, and 4. The default is `+Onosize`.

Using software development tools

3

This chapter provides an overview of some tools that assist in program development. These tools include utility programs, debuggers, and profilers.

Some of the programs discussed in this chapter are optional. If you are unsure whether a program is installed on your system, check with your system manager.

The following utility program makes software development easier:

- `make` — makes program compilation easier and eliminates redundant compilations.

The debugger utility is:

- CXdb — a window-oriented symbolic debugger for code that has been compiled to run on Exemplar systems.

The profiling utilities are:

- CXpa — a window-oriented performance profiling utility for code that has been compiled to run on Exemplar systems.
- `cxoi` — a command-line utility for profiling object files and archive libraries that have not been compiled with the `+pa` option flag. For more information, refer to the `cxoi` man page.

Program development utility

The utility described in this section helps you in creating and managing C++ programs.

make utility

You should use `make` when you have several files that compose a program and you do not need to recompile all of them. The `make` utility uses the time and date stamps of source and object files to decide whether a file must be compiled. It compiles only:

- Those files that have been modified since you last compiled the program
- Any files that depend on the changed files

Thus, if a program depends on twelve separate compilation units but only two of them have been modified, `make` compiles only those two before the entire program is linked.

Consider the sample make file shown in Figure 3.

```
#                                                    1
# Make file for a short example.                     2
#                                                    3
myprog: myprog.o second.o                      #     4
   CC myprog.o second.o -o myprog              #     5
                                               #     6
myprog.o: myprog.C local.H                     #     7
   CC -c myprog.C                              #     8
                                               #     9
second.o: second.C local.H pivot.H             #     10
   CC -c second.c                              #     11
```

Figure 3 Sample make file

If this file is stored in a file named makefile, the executable program myprog can be created by entering the command `make`.

The first three lines contain comments; comment lines start with a # character. Line 7 states that myprog.o is dependent on the two files, myprog.C and local.H. If one or both of these are modified after myprog.o is created, the command on line 8 is executed. This line creates an up-to-date version of myprog.o. The remaining lines follow the same format. Consequently, if pivot.H is changed, commands on line 11 and line 5 are executed to create an executable file, myprog.

This example, while useful enough for small programs, uses only a few of the features that the `make` utility provides. You can find information on the additional features in the make(1) man page.

Debugger utilities

The following section describes the utility programs you can use to debug a program.

CXdb debugger

Exemplar CXdb is a window-based debugger that has all the debugging functions found in traditional debuggers. To generate the information necessary for using CXdb, you must compile your program with one of the following options:

- `-g`
- `-g1`

CXdb can perform these functions:

- Debug program source code or disassembled code
- Debug programs containing multiple source modules
- Access program variables by name
- Provide debugging contexts for source code and disassembled code in a windowing environment
- Attach to a running process
- Execute debugger commands while your process is running
- Debug parallel applications
- Create aliases and macros to simplify debugger commands

The CXdb windowing environment supports line-oriented terminals and workstations capable of supporting X displays. This windowing environment eases the task of debugging programs that contain multiple threads of execution. Refer to the *CXdb Reference* for additional information on CXdb.

Using CXdb on a C++ program

To use CXdb in X window mode on a C++ program, follow these steps:

Step 1 Set your `DISPLAY` environment variable (if it is not already set). For example, in C shell, enter:

```
setenv DISPLAY mydisplay:0.0
```

Step 2 Compile and link your program with the `-g` option:

```
CC -g prog.C
```

Step 3 Invoke CXdb on the executable file:

```
cxdb a.out &
```

Refer to the *CXdb Reference* for additional information on CXdb.

Profiler utility

Profilers help you to analyze and improve the performance of your programs.

The Exemplar C++ compiler supports the CXpa profiler on SPP-UX V5.0 or higher running on an SPP1200, SPP1600, S2000 server, or X2000 server. CXpa is an optional product, and it is described in detail in the *CXpa Reference*.

CXpa is an interactive tool that can monitor program activity at the routine level and the loop level. Before using CXpa, you need to compile and link your C++ code with the `+pa` compiler option to add instrumentation, or additional code, for CXpa to read.

The following list shows metrics that CXpa can capture for different platforms. For more information about these CXpa metrics, see the *CXpa Reference*.

- All Exemplar and SPP Series platforms
  - Execution counts
  - CPU time
  - Wall clock time
  - Concurrency factor (CPU time/Wall clock time)
  - Dynamic call graph
- Exemplar S-Class and X-Class servers, and SPP 1600 Series
  - Locally resolved cache miss counts and latency
  - Remotely resolved cache miss counts and latency
  - Locally and remotely resolved (total) cache miss counts and latency
  - Average cache miss latency and other derived metrics
- SPP 1200 and SPP 1600 Series
  - Data cache misses, accesses, and latency
  - Data cache hit rates
  - Average data cache miss latency
  - Instruction cache misses and latency
  - Average instruction cache miss latency
  - Instructions completed
  - Clock cycles
  - Data and instruction TLB (translation lookaside buffer) misses

Using CXpa on a C++ program

To use CXpa in GUI mode on a C++ program, follow these steps:

Step 1 Set your `DISPLAY` environment variable (if it is not already set). For example, in C shell, enter:

```
setenv DISPLAY mydisplay:0.0
```

Step 2 Compile and link your program with the `+pa` option (see "Compiler options" on page 10):

```
CC +pa prog.C
```

Step 3 Invoke CXpa on the executable file:

```
cxpa a.out &
```

Refer to the *CXpa Reference* for additional information on CXpa.

Inter-language communication

4

This chapter presents guidelines for linking Exemplar C++ modules with modules written in C or Fortran on Exemplar systems and HP 9000 Series 700 systems.

Introduction

A C++ *module* is a file containing one or more variable or function declarations, one or more function definitions, or similar items logically grouped together. Mixing modules written in C++ with modules written in C is relatively straightforward because C++ is essentially a superset of C. Mixing C++ modules with modules written in languages other than C is more complicated.

When you create an executable file from a set of programs written in different languages including C++, you must observe the following guidelines:

- In general, you must write the overall control of the program in C++. In other words, the `main()` function must appear in a C++ module.
- You must follow the case-sensitivity conventions for function names in each of the languages.
- You must make sure that the data types in the different languages correspond. Do not mismatch data types for parameters and return values.
- You must follow the different storage layouts for aggregates in each of the languages.
- You must use the `extern "C"` linkage specification to declare any modules that are not written in C++. This is true whether or not the module is written in C.

- You must use the `extern "C"` linkage specification to declare any modules that are written in C++ and called from other languages.

Data compatibility between C and C++

Since C++ is a superset of C, many of the data types are identical. The two languages have the identical primitive types `char`, `short`, `int`, `long`, `float`, and `double`. ANSI C and C++ also support a `long double` type. Pointers, structures, and unions that can be declared in C are also compatible. Arrays composed of any of these types are compatible.

C++ classes are generally incompatible with C structures. The following features of the C++ class facility may cause the compiler to generate extra code, extra fields, or extra tables:

- Multiple visibility of members (that is, having both `private` and `public` members in a class)
- Single or multiple inheritance
- Virtual functions

It is the use of these features, rather than the use of the `class` keyword instead of the `struct` keyword, that introduces incompatibility with C structures.

Calling C modules from C++

Since C++ is a superset of C, C functions can be called from C++ programs. The following differences, however, must be taken into account:

- You must use the `extern "C"` linkage specification to declare the C functions.
- You need to account for the differences in argument passing conventions between C and C++.
- You must write the overall control of the program in C++.

Using the `extern "C"` linkage specification

To handle overloaded function names, the C++ compiler generates unique names for all functions declared in a C++ program. The compiler uses an implementation-dependent function-name encoding scheme to create these names. A linkage directive tells the compiler to inhibit the default encoding of a function name for a particular function.

If you intend to call a C function from a C++ program, you must tell the compiler not to use its usual encoding scheme when you

declare the C function. If the names do not match, the linker cannot resolve them. To avoid linkage problems, use a linkage directive when you declare the C function in the C++ program.

Exemplar C++ linkage directives have one of the following formats:

```
extern "C" function_declaration
```

or

```
extern "C"
      {
      function_declaration1
      function_declaration2
      ...
      function_declarationN
      }
```

For example, the following declarations are equivalent:

```
extern "C" char* get_name(); // external C module
```

and

```
extern "C"
      {
      char* get_name(); // external C module
      }
```

Differences in argument passing conventions

By default, the Exemplar C++ compiler does not generate function prototypes in the C code it creates in translator mode. As a result, the Exemplar C compiler applies the argument widening rules of C without prototyping. This means that `char` and `short` types are promoted to `int`, and `float` is promoted to `double`.

In programs written entirely in C++, this does not cause any problem because the arguments are consistently handled within the program. However, if your C++ code calls functions written in C, you must make sure that the called C functions do not use function prototypes that suppress argument widening. If they do, your C++ code may pass wider arguments than your C code is expecting.

In translator mode you can use the `+a1` option with `CC` to tell the translator to emit function prototypes in the C code it generates. When you use `+a1`, the linker links in the ANSI version of libC.a, which is named libC.ansi.a.

In compiler mode, the `+a0` option causes parameters of type `float` to be promoted to type `double`. When the `+a1` option is used in

compiler mode, `float` parameters are not promoted but are passed as type `float`.

The `main()` function

In general, the overall control of a program with mixed C and C++ modules must be written in C++. This means that the `main()` function must appear in a C++ module rather than in a C module. There are two exceptions:

- Programs without any global class objects containing constructors or destructors
- Programs without static objects

The following C++ program, calling_c.C, calls a C function `get_name()`.

```
//*************************************************************************
// Program name is calling_c.C                                            *
//*************************************************************************
// This is a C++ program that illustrates calling a function written in C.  *
// It calls the get_name() function, which is in the get_name.c module.  The *
// object modules generated by compiling the calling_c.C module and by       *
// compiling the get_name.c module must be linked to create an executable    *
// file.                                                                     *
//*************************************************************************
#include <stream.h>
#include "string.h"
//*************************************************************************
extern "C" char* get_name();   // declare the external C module
class account
{
private:
     char* name;               // owner of the account
protected:
     double balance;           // amount of money in the account
public:
     account(char* c)          // owner of the account
          {
               name = new char [strlen(c) +1];
               strcpy(name,c);
               balance = 0;
          }
     void display()
          {
               cout << name << " has a balance of " << balance << "\n";
          }
};
```

```
main()
{
     account* my_checking_acct = new account (get_name());
     my_checking_acct->display();
             // send a message to my_checking_account to display itself
}
```

The following example shows the C module get_name.c. This function is called by the C++ program in the previous example.

```
/*****************************************************************************
*  This is a C function that is called by main() in a C++ module calling_c.C. *
*  The object modules generated by compiling this module and by compiling the *
*  calling_c.C module must be linked to create an executable file.            *
*****************************************************************************
*/
#include <stdio.h>
#include "string.h"
char* get_name()
{
     static char name[80];
     printf("Enter the name: ");
     scanf("%s",name);
     return name;
}
/*****************************************************************************/
```

To compile the program calling_c.C and the get_name.c function into an executable named `balance` in translator mode on an Exemplar system, you would use the following commands:

```
CC -c +a1 calling_c.C
cc -c get_name.c
CC -o +a1 balance calling_c.o get_name.o
```

You must use the `CC` driver for the link phase because `CC` performs several functions that support the C++ constructs during the link phase. Linking programs that use classes with the C compiler driver `cc` causes unpredictable results at run time.

Calling C++ modules from C

It is possible under some circumstances to call C++ functions from C programs. In order for this to work, your code must meet the following constraints:

- To prevent a C++ function name from being mangled, the function definition and all declarations used by the C++ code must use `extern "C"`.
- The C program must include a call to the function `_main` as the first executable statement in `main()`. Object libraries require this because `_main` calls the static constructors to initialize the libraries' static data items.
- Member functions of classes in C++ are not callable from C. If a member function routine is needed, a nonmember function in C++ can be called from C which in turn calls the member function.
- Since the C program cannot directly create or destroy C++ objects, it is the responsibility of the writer of the C++ class library to define interface routines that call constructors and destructors, and it is the responsibility of the C program to call these interface routines to create such objects before using them and to destroy them afterwards.
- The C program must not define an equivalent `struct` definition for the class definition in C++. The class definition may contain bookkeeping information that is not guaranteed to work on every architecture. All access to members should be done in the C++ module.

The following C++ module is written according to the above constraints so that it can be called by a C program.

```
//*****************************************************************************
// C++ obj_ptr.C module that manipulates an object obj                        *
//*****************************************************************************
#include <stream.h>

typedef class obj* obj_ptr;
extern "C" void initialize_obj (obj_ptr& p);
extern "C" void delete_obj (obj_ptr p);
extern "C" void print_obj (obj_ptr p);

struct obj {
private:
     int x;
public:
     obj() {x = 7;}
     friend void print_obj(obj_ptr p);
};
```

```
//C interface routine to initialize an object by calling the constructor
void initialize_obj(obj_ptr& p)
{
     p = new obj;
}

//C interface routine to destroy an object by calling the destructor.
void delete_obj(obj_ptr p)
{
     delete p;
}
//C interface routine to display manipulating the object.
void print_obj(obj_ptr p)
{
     cout << "the value of object->x is " << p->x << "\n";
}
```

The following C program calls the obj_ptr.C module to manipulate an object.

```
/*****************************************************************************
* C program call_obj.c demonstrates an interface to the C++ module obj_ptr.C. *
* This application needs to be linked using the CC driver.                     *
*****************************************************************************/

typedef struct obj* obj_ptr;

main () {
     /* C++ object. None of the routines should try to manipulate the fields. */

     obj_ptr f;

    /*
     *  The first executable statement needs to be a call to _main so the
     *  the static constructors will be created in libraries that have
     *  constructors defined. In this application, the stream library
     *  contains data elements that match the conditions.
     */
     _main();

    /*
     * Initialize the data object.  Taking the address of f is compatible
     * with the C++ reference construct.
     */
     initialize_obj(&f);

     /* Call the routine to manipulate the fields */
     print_obj(f);

     /* Destroy the data object */
     delete_obj(f);
}
```

To compile the program call_obj.c and the obj_ptr.C module into an executable named `example` on an Exemplar system, you would use the following commands:

```
CC -c obj_ptr.C
cc -c call_obj.c
CC -o example call_ptr.o obj_ptr.o
```

You must use the `CC` driver for the link phase because `CC` performs several functions that support the C++ class mechanism during the link phase. Linking programs that use classes with the C compiler driver `cc` causes unpredictable results at run time.

Calling Fortran modules from C++

It is possible to mix C++ modules with modules written in Exemplar Fortran or standard HP Fortran. In general, the overall control of the program must be written in C++. In other words, the `main()` function must appear in a C++ module.

Passing arguments by reference

There are two methods of passing arguments, by reference or by value. Passing by reference means that the routine passes the address of the argument rather than the value of the argument.

When calling Fortran functions from C++, you must ensure that the calling function and the called function use the same method of argument passing for each individual argument. When calling external functions in Fortran, you need to know the calling convention for the order of arguments.

You should not pass structures or classes to Fortran routines. For maximum compatibility and portability, only simple data types should be passed to routines. All C and C++ parameters are passed by value except arrays and functions, which are passed as pointers.

Fortran passes all arguments by reference. This means that all actual parameters in a Exemplar C++ call to a Fortran routine must be pointers or variables prefixed with the unary address operator `&`.

The simplest way to reconcile the differences in argument-passing conventions is to use reference variables in your C++ code. Declaring a parameter as a reference variable in a prototype causes the compiler to pass the argument by reference when the function is invoked. The following C++ code example illustrates a reference variable.

```
int main(void)
{
     extern void pas_func(short &); // declare a reference variable
     short x;
        .
        .
        .
     pas_func(x);                   // pas_func must accept parameters
        .                           // by reference
        .
        .
}
```

See Chapter 8, "Declarators," of *The Annotated C++ Reference Manual* for more information about using reference variables.

Using `extern "C"` linkage

Whenever you mix C++ modules with Fortran modules, make sure that you use the `extern"C"` linkage to declare any C++ functions that are called from a non-C++ module and to declare the Fortran routines. See "Using the extern "C" linkage specification" on page 36 for more information about using the `extern"C"` linkage specification.

Strings

C++ strings are not compatible with Fortran strings. C++ strings are null terminated, but Fortran strings are not. Also, strings are passed as string descriptors in Fortran. The descriptor consists of the address of the character string followed by the value of the string length.

Arrays

Exemplar C++ stores arrays in row-major order, while Fortran stores arrays in column-major order. The lower bound for arrays in C++ is 0, and the default lower bound for standard HP Fortran and Exemplar Fortran is 1.

Definitions of TRUE and FALSE

The definitions of TRUE and FALSE differ between programming languages and between computer architectures. In Fortran, TRUE and FALSE are the values for the LOGICAL type. C and C++ do not have a LOGICAL type; these languages use integers instead.

In C and C++ on both Exemplar SPP systems and HP 9000 Series 700 systems, FALSE has a value of 0 and TRUE can have any nonzero integer value.

In Fortran on both Exemplar SPP systems and HP 9000 Series 700 systems, FALSE has a value of 0 and TRUE has a value of 1, unless explicitly changed using the `+e` or `+E2` option.

File references

Fortran I/O routines require a logical unit number to access a file, while C++ accesses files using a stream pointer.

A Fortran logical unit cannot be passed to a C++ routine to perform I/O on the associated file. Likewise, a C++ file pointer cannot be used by a Fortran routine. However, a file created by a program in either language can be used by a program of the other language if the file is opened within a program of that language. In addition, you can implement I/O in Fortran programs in a way that is compatible with C++ by using stream I/O instead of Fortran I/O. Refer to your Fortran manual for information on using stream I/O.

Linking Fortran routines with C++ programs

When you link Fortran routines with a C++ program, you must use the `-lcl` and `-lisamstubs` options on the `CC` command line. For example, to compile and link a Fortran routine fort_call.f with a C++ program called prog.C to create an executable called `mixed`, you would use the following commands:

```
CC -c prog.C
fc -c fort_call.f
CC -o mixed -lcl -lisamstubs prog.o fort_call.o
```

Developing parallel C++ applications

5

This chapter describes methods for creating parallel C++ applications for Exemplar S-Class and X-Class servers.

Parallel programming overview

One method of solving large computational problems is to divide the problem into several small tasks. The individual tasks are distributed among the processing resources available on a computer system.

The architecture of Exemplar S-Class and X-Class servers is well suited for parallel programming. Multiple processors with large caches, large amounts of physical memory, and high-speed communications between processors allow parallel programs to execute efficiently. For more information about parallel programming on Exemplar systems, refer to the *Exemplar Programming Guide.*

The Exemplar C++ compiler is capable of performing some code parallelizations automatically. See "Optimization options" on page 21 for more information about compiler optimization options.

In addition to the automatic parallelizations, Exemplar systems support two programming models for explicit parallelization: *message-passing programming* and *shared-memory programming*. Each of these programming models is supported by its own programming library.

Message-passing programming

In the message-passing programming model, parallel tasks communicate with each other by means of *messages*. Messages are implemented by system library calls that package, transmit, and receive data.

Exemplar systems support two types of message passing programming through the following libraries:

- HP PVM — an implementation of the Parallel Virtual Machine (PVM) programming model on Exemplar systems. For more information on using PVM, refer to the *HP PVM User's Guide* and the book *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing.*
- HP MPI — an implementation of the Message Passing Interface (MPI) on Exemplar systems. For more information on using MPI, refer to the *HP MPI User's Guide* and the book *MPI: The Complete Reference.*

Note

The current versions of HP PVM and HP MPI support standard C function calls only.

This chapter assumes a familiarity with either PVM or MPI, and is therefore concerned primarily with C++ examples.

Notes on running PVM applications

This section provides a brief overview of how to run a HP PVM application on an Exemplar system. For a detailed description, refer to the *HP PVM User's Guide.*

All C++ programs that use the HP PVM library must contain the following line:

```
#include "pvm3.h"
```

The pvm3.h header file contains the type and function definitions for the HP PVM library. This file is located in the /opt/pvm3/include directory.

To initialize PVM on an Exemplar system, perform the following steps:

1. Create your PVM hostfile. The simplest hostfile you can use on an Exemplar system contains the following line:

   ep=*dir*:*dir*...

   where *dir*:*dir*. . . is a list of directories that PVM searches for executables. Your executable program must reside in one of these directories.

2. Set the PVM_ARCH and PVM_ROOT environment variables, and add the directory containing PVM executables to your path. In C shell notation, enter the following:

```
setenv PVM_ARCH CSPP
setenv PVM_ROOT /opt/pvm3
set path=($path /opt/pvm3/lib)
```

3. Start the PVM daemon by entering the following command:

```
/opt/pvm3/lib/CSPP/pvmd3 hostfile &
```

where *hostfile* is the name of your hostfile. You must specify the path to the hostfile if it is not in the current working directory.

The current working directory for spawned PVM processes is your home directory by default. The spawned processes will look for input files in that directory and will write output files to that directory, not the directory where the program is located. You can change the current working directory for PVM applications by adding a wd= entry to your PVM hostfile.

Notes on running MPI applications

This section provides a brief overview of how to run an MPI application on an Exemplar system. For a detailed description, refer to the *HP MPI User's Guide*.

All C++ programs that use the HP MPI library must contain the following line:

```
#include "mpi.h"
```

The mpi.h header file contains the type and function definitions for the HP MPI library. This file is located in the /opt/mpi/include directory.

To initialize HP MPI on an Exemplar system, perform the following steps (C Shell notation):

1. Add the MPI binaries to your path:

```
set path=($path /opt/mpi/bin)
```

2. Set the optional environment variables MPI_TOPOLOGY, MPI_GLOBMEMSIZE, and MPI_FLAGS as desired.
3. Start your application program using the mpirun utility:

```
mpirun -np n executable
```

where *n* is the number of MPI processes to use, and *executable* is the name of your program executable file.

Shared-memory programming (CPSlib)

Exemplar systems support a shared-memory programming model that eliminates the overhead involved in passing messages between threads. The fact that physical memory in Exemplar systems is globally shared among all processors in the system allows efficient control of data and tasks in a parallel application. For more information, see the *Exemplar Programming Guide.*

Exemplar C++ supports shared memory programming through the Exemplar CPSlib library. This chapter assumes a familiarity with CPSlib and is therefore concerned primarily with C++ examples. For more information on using CPSlib, refer to the *Exemplar Programming Guide.*

All C++ programs that use the Exemplar CPSlib library must contain the following line:

```
#include <cps.h>
```

The cps.h header file contains the type and function definitions for the Exemplar CPSlib library. This file is located in the /usr/include directory.

A program that uses CPSlib must be linked for parallel execution. You must use one of the following methods to create a CPSlib shared-memory application.

Creating a shared-memory application using CPSlib

You can create a shared-memory application that uses CPSlib in either of two ways:

- The easiest way to link to CPSlib is to use the `+O3` and `+Oparallel` options on the `CC` command line:

  `CC +O3 +Oparallel` *program_name*

- If you are not compiling your application at level `+O3` or higher, you must explicitly link to CPSlib by using the following `CC` command options, in the order listed here:

`CC -lpthread -lcps -lpthread -lail -lcnx_syscall` *program_name*

Creating a shared-memory application using the `mpa` command

You can use the Exemplar `mpa` command to modify an executable program to run as a parallel application. For example, to make the executable program CpsSort parallel, use the following command:

```
mpa -m -n -parallel -min min_threads CpsSort
```

where *min_threads* is the minimum number of threads that the CpsSort executable can spawn. The `-n` parameter prevents `mpa` from executing the program.

There are additional attributes you can specify for an executable program using `mpa`; for more information, see the mpa(1) man page.

A parallel programming example

The following sections show how to convert an example C++ program for parallel execution on an Exemplar system.

The example program is first presented with no explicit parallelization. Next, the program is restructured to a pseudo-parallel form that allows the main problem (an integer sort) to be divided into tasks (threads). Finally, message-passing (PVM) and shared-memory (CPSlib) versions of the restructured program are presented.

An odd-even sort program

This program implements an odd-even transposition sort algorithm. The program takes a list of integers and prints a sorted list of those integers.

The compile and link commands for this example program are:

```
CC -c SerialEntry.C
CC -c Serial.C
CC -o serial_example SerialEntry.o Serial.o
```

Unoptimized program Serial.C

The first version of the program uses a `for` loop that examines adjacent pairs of numbers in the list and exchanges them if the number on the left is greater than the number on the right. The loop passes through the list examining the odd entries and their right neighbors, then examines the even entries and their right neighbors. The loop executes $n/2$ times, where n is the number of entries in the input list.

```
#include <stdio.h>          // for sprintf
#include <stdlib.h>         // for atoi
#include <iostream.h>
#include "SerialEntry.h"

main(int argc, char **argv)
{

   if (argc != 3) {
      cout << "Usage: "
           << argv[0]
           << " <filename> <number of elements>"
           << endl;
      exit(1);
   }
   int numEntries = atoi(argv[2]);

   char outfname[256];
   sprintf(outfname, "%s_", argv[1]); // argv[1] : input filename

   BlockOfEntries *aBlock = new BlockOfEntries(argv[1], outfname);


   for(int j=0;j<numEntries/2;j++) {
     aBlock->singleStepOddEntries();
     aBlock->singleStepEvenEntries();
  }

  delete aBlock; // during destruction, output is written to outfname
}
```

SerialEntry.h

This header file defines the classes for the basic odd-even sort program. The `Entry` class represents a single entry in the list of entries to be sorted. In this case, the `Entry` class encapsulates an integer. This encapsulation allows the same classes and sort algorithm to be used for other types of entities. The `BlockOfEntries` class represents the entire list of entries. At the time it is destroyed, it contains a sorted version of the list.

```
#include <iostream.h>
#include <fstream.h>
class Entry {
    private:
       int value;
    public:
       Entry()
           { value = 0; }
       Entry(int x)
           { value = x; }
       Entry(const Entry &e)
           { value = e.getValue(); }
       Entry& operator= (const Entry &e)
           { value = e.getValue(); return (*this); }
       int getValue() const
           { return value; }
       int operator> (const Entry &e) const
           { return (value > e.getValue()); }
       friend ostream& operator<< (ostream &os, const Entry &e)
           { return (os << e.value); }
       friend istream& operator>> (istream &is, Entry &e)
           { return (is >> e.value); }

};

class BlockOfEntries {
    private:

       Entry **entries;
       int numOfEntries;

       ifstream inFile;
       ofstream outFile;

    public:

       BlockOfEntries(char *infname, char *outfname);
       ~BlockOfEntries();
       void singleStepOddEntries();
       void singleStepEvenEntries();

};
```

SerialEntry.C

This file defines methods for the classes defined in SerialEntry.h.

```
#include "SerialEntry.h"

BlockOfEntries::BlockOfEntries(char *infname, char *outfname) :
    inFile(infname, ios::in), outFile(outfname, ios::out)

{
   inFile >> numOfEntries; // read the number of entries
   entries = new Entry *[numOfEntries]; // allocate space for entries
   for(int i=0;i<numOfEntries;i++) { // read all the entries
      entries[i] = new Entry; // use default constructor
      inFile >> *(entries[i]); // use overloaded global operator>>
   }

}
BlockOfEntries::~BlockOfEntries()
{

  // output number of entries and the sorted entries; delete  them too
  outFile << numOfEntries << endl;
  for(int i=0;i<numOfEntries;i++) {
    outFile << *(entries[i]) << endl; // use overloaded global operator<<
    delete entries[i];
  }
  // delete array
  delete [] entries;
}

void BlockOfEntries::singleStepOddEntries()
{
  for(int i=1;i<numOfEntries-1;i+=2) {
    if (*(entries[i]) > *(entries[i+1]) ) { // use Entry::operator>
       Entry *temp = entries[i+1];
       entries[i+1] = entries[i];
       entries[i] = temp;
    }
  }

}

void BlockOfEntries::singleStepEvenEntries()
{
  for(int i=0;i<numOfEntries-1;i+=2) {
    if (*(entries[i]) > *(entries[i+1]) ) { // use Entry::operator>
       Entry *temp = entries[i+1];
       entries[i+1] = entries[i];
       entries[i] = temp;
    }
  }
}
```

A pseudo-parallel version of the sort program

This version of the basic sort program contains the structures needed for parallel execution. The approach is to divide the main `for` loop into a number of threads, each operating on a separate `BlockOfEntries`. In order to compare entries across the boundaries of blocks, the `setRightShadow` and `setLeftShadow` functions have been added to create copies of the entries on the boundaries of blocks.

Although this version of the program contains the necessary structures for parallelization, it simulates parallel execution using `for` loops instead of parallel threads.

The compile and link commands for this example program are:

```
CC -c pseudo_parallel.C
CC -c Entry.C
CC -o pseudo_parallel pseudo_parallel.o Entry.o
```

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "Entry.h"

main(int argc, char **argv)
{

   if (argc != 4) {
      cout << "Usage: "
           << argv[0]
           << " <base filename> <number of elements> <number of threads>"
           << endl;
      exit(1);
   }
   int numEntries = atoi(argv[2]);
   int numThreads = atoi(argv[3]);

   // Every thread has a block of entries
   BlockOfEntries **aBlock = new BlockOfEntries*[numThreads];

   // each thread opens its file and reads in the entries
   char infname[256], outfname[256];
   for(int i=0;i<numThreads;i++) {
      sprintf(infname, "%s%d", argv[1], i);
      sprintf(outfname, "%s_%d", argv[1], i);
      aBlock[i] = new BlockOfEntries(infname, outfname);
   }
```

```
    for(int j=0;j<numEntries/2;j++) {

      int left, right;

      // each thread gets its shadow entries from neighbors
      for(i=0;i<numThreads;i++) {
        if (i==0) {
           right = 1;
           Entry rightVal = aBlock[right]->getLeftEnd();
           aBlock[i]->setRightShadow(rightVal);
        }
        else if (i==numThreads-1) {
           left = numThreads - 2;
           Entry leftVal = aBlock[left]->getRightEnd();
           aBlock[i]->setLeftShadow(leftVal);
        }
      else {
           left = i-1;
           right = i+1;
           Entry leftVal = aBlock[left]->getRightEnd();
           aBlock[i]->setLeftShadow(leftVal);
           Entry rightVal = aBlock[right]->getLeftEnd();
           aBlock[i]->setRightShadow(rightVal);       }

      }

      for(i=0;i<numThreads;i++) {
         aBlock[i]->singleStepOddEntries();
      }


      for(i=0;i<numThreads;i++) {
         aBlock[i]->singleStepEvenEntries();
      }
    }

    for(i=0;i<numThreads;i++)
      delete aBlock[i];
    delete [] aBlock;
}
```

Class definitions for the parallel program

The Entry.h and Entry.C files define the classes and methods for the pseudo-parallel and parallel (PVM and CPSlib) versions of the sort program.

Entry.h file

This header file defines the classes for the pseudo-parallel and parallel versions of the sort program. The `BlockOfEntries` class has been modified from the original version to include the left and right shadow entries (`setLeftShadow` and `setRightShadow` functions).

```
#include <iostream.h>
#include <fstream.h>

class Entry {
    private:
       int value;
    public:
       Entry()
           { value = 0; }
       Entry(int x)
           { value = x; }
       Entry(const Entry &e)
           { value = e.getValue(); }
       Entry& operator= (const Entry &e)
           { value = e.getValue(); return (*this); }
       int getValue() const            { return value; }
       int operator> (const Entry &e) const
           { return (value > e.getValue()); }
       friend ostream& operator<< (ostream &os, const Entry &e)
           { return (os << e.value); }
       friend istream& operator>> (istream &is, Entry &e)
           { return (is >> e.value); }
};

class BlockOfEntries {
    private:

       Entry **entries;
       int numOfEntries;

       ifstream inFile;
       ofstream outFile;
```

```
    public:

        BlockOfEntries(char *infname, char *outfname);        ~BlockOfEntries();

        void setLeftShadow(const Entry &e)
            { *(entries[0]) = e; }
        void setRightShadow(const Entry &e)
            { *(entries[numOfEntries-1]) = e; }

        const Entry& getLeftEnd()
            { return *(entries[1]); }
        const Entry& getRightEnd()
            { return *(entries[numOfEntries-2]); }

        void singleStepOddEntries();
        void singleStepEvenEntries();
};
```

Entry.C file

This file defines methods for the classes defined in Entry.h. This file applies to the parallel and pseudo-parallel versions of the sort program. In this version of the file, the `entries` array is allocated with extra space for the shadow entries, and the `~BlockOfEntries()` function deletes the shadow entries along with the other entries.

```
#include "Entry.h"
#include <limits.h>

const Entry MAXENTRY(INT_MAX);
const Entry MINENTRY(INT_MIN);

BlockOfEntries::BlockOfEntries(char *infname, char *outfname) :
    inFile(infname, ios::in), outFile(outfname, ios::out)

{
   inFile >> numOfEntries; // read the number of entries
   numOfEntries += 2; // add left and right shadow entries
   entries = new Entry *[numOfEntries]; // allocate space for entries
   for(int i=1;i<numOfEntries-1;i++) { // read all the entries
      entries[i] = new Entry; // use default constructor
      inFile >> *(entries[i]); // use overloaded global operator>>
   }

   // initialize shadow entries
   entries[0] = new Entry(MINENTRY);
   entries[numOfEntries-1] = new Entry(MAXENTRY);
}
```

```
BlockOfEntries::~BlockOfEntries()
{

  // output number of entries and the sorted entries; delete them too
  outFile << numOfEntries-2 << endl;
  for(int i=1;i<numOfEntries-1;i++) {
    outFile << *(entries[i]) << endl; // use overloaded global operator<<
    delete entries[i];
  }
  // delete shadows and array
  delete entries[0];
  delete entries[numOfEntries-1];
  delete [] entries;
}

void BlockOfEntries::singleStepOddEntries()
{
  for(int i=1;i<numOfEntries-2;i+=2) {
    if (*(entries[i]) > *(entries[i+1]) ) { // use Entry::operator>
       Entry *temp = entries[i+1];
       entries[i+1] = entries[i];
       entries[i] = temp;
    }
  }

}

void BlockOfEntries::singleStepEvenEntries()
{
  for(int i=2;i<numOfEntries-2;i+=2) {
    if (*(entries[i]) > *(entries[i+1]) ) { // use Entry::operator>
       Entry *temp = entries[i+1];
       entries[i+1] = entries[i];
       entries[i] = temp;
    }
  }
}
```

A message-passing implementation

This example uses HP PVM library calls to implement a parallel version of the odd-even sort program. This program uses a master-slave model in which a master task spawns a slave task for each input file. The slave tasks are assigned indices from 0 to M-1, where M is the number of input files.

The master program passes the task ID of left and right neighbors to each slave task so that the tasks can pass shadow values (the values of the entries on the border of each `BlockOfEntries`) to each other. The exchange of shadow values is done using the `pvm_send` and `pvm_receive` functions. The integer embedded in an `Entry` is packed into the send buffer using the `pvm_pkint` function and then sent to the appropriate neighbor task. A blocking `pvm_recv` is used to receive shadow values.

The slave tasks are formed into a group using the `pvm_joingroup` function. This allows the slaves to synchronize using barriers.

The compile and link commands for this example program are:

```
CC -c -I/opt/pvm3/include pvm_example.C
CC -c -I/opt/pvm3/include Entry.C
CC -o pvm_example pvm_example.o Entry.o -L/opt/pvm3/lib/CSPP -lgpvm3 -lpvm3
   -lail -lcnx_syscall
```

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "Entry.h"
#include "pvm3.h"


main(int argc, char **argv)
{

   if (argc != 4) {      cout << "Usage: "
           << argv[0]
           << " <base filename> <number of elements> <number of tasks>"
           << endl;
      exit(1);
   }

   int numEntries = atoi(argv[2]);
   int numTasks = atoi(argv[3]);

   int myId = pvm_mytid();
   int pId = pvm_parent();

   int val, myIndex, leftId, rightId;
```

```
if (pId == PvmNoParent) { // Master

  // spawn slave tasks
  char *slaveArgs[4];

  slaveArgs[0] = argv[1];
  slaveArgs[1] = argv[2];
  slaveArgs[2] = argv[3];
  slaveArgs[3] = '\0';;
  int *slaveTids = new int[numTasks];
  pvm_spawn(argv[0], slaveArgs, PvmTaskDefault, "", numTasks, slaveTids);

  // inform each slave about its index and its neighbor's ids
  for(int i=0;i<numTasks;i++) {
     pvm_initsend(PvmDataDefault);
     pvm_pkint(&i, 1, 1); // slave x's index
     leftId = (i!=0) ? slaveTids[i-1] : 0; // x's left slave's id
     pvm_pkint(&leftId, 1, 1 );
     rightId = (i!=numTasks-1) ? slaveTids[i+1] : 0; // right slave's id;
     pvm_pkint(&rightId, 1, 1);
     pvm_send(slaveTids[i], 0);
  }

  // wait for slaves to complete sorting
  for(i=0;i<numTasks;i++) {
     pvm_recv(slaveTids[i], 0);
  }
}
else { // Slave task

  // receive info from master
  pvm_recv(pId, 0);
  pvm_upkint(&myIndex, 1, 1);
  pvm_upkint(&leftId, 1, 1);
  pvm_upkint(&rightId, 1, 1);

  // join a group
  pvm_joingroup("worker");

  char infname[256], outfname[256];

  sprintf(infname, "%s%d", argv[1], myIndex);
  sprintf(outfname, "%s_%d", argv[1], myIndex);
```

```
    // read in the block of entries
    BlockOfEntries *aBlock = new BlockOfEntries(infname, outfname);
    for(int j=0;j<numEntries/2;j++) {
      // synchronize before updating shadow entries
      pvm_barrier("worker", numTasks);


      // update shadow entries
      // everyone except 0 sends leftEnd to left
      if (myIndex != 0) {
           val = aBlock->getLeftEnd().getValue();
           pvm_initsend(PvmDataDefault);
           pvm_pkint(&val, 1, 1);
           pvm_send(leftId, 0);
      }
      // everyone except numTasks-1 receives rightShadow from right
      if (myIndex != numTasks-1) {
           pvm_recv(rightId, 0);
           pvm_upkint(&val, 1, 1);
           aBlock->setRightShadow(Entry(val));
      }
      // everyone except numTasks-1 sends rightEnd to right
      if (myIndex != numTasks-1) {
           val = aBlock->getRightEnd().getValue();
           pvm_initsend(PvmDataDefault);
           pvm_pkint(&val, 1,1);
           pvm_send(rightId, 0);
      }
      // everyone except 0 receives leftShadow from left
      if (myIndex != 0) {
           pvm_recv(leftId, 0);
           pvm_upkint(&val, 1, 1);
           aBlock->setLeftShadow(Entry(val));
      }

      aBlock->singleStepOddEntries();
      aBlock->singleStepEvenEntries();
    }

    delete aBlock; // output sorted entries and delete the block

    // inform the master that the sorting is complete
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&val, 1,1);
    pvm_send(pId, 0);
   }

   pvm_exit();
}
```

A shared-memory implementation

This version of the program is a parallel implementation using CPSlib library calls. As in the PVM version, a thread is spawned for each input file. The blocks of entries are allocated in global shared memory (GSM) and are therefore readable and writable by all threads. The threads follow the "owner-computes" rule — only the thread that is assigned as the owner of a block writes to that block.

The shared-memory implementation allows each thread to update the values of its shadow entries by reading the values of neighbor block entries from the global shared memory. This is more efficient than the PVM implementation, where the threads must pass these values to each other in messages.

Synchronization of the threads is accomplished by using the `cps_barrier` function to implement a shared-memory barrier.

The compile and link commands for this example program are:

```
CC -c -I/opt/pvm3/include gsm_example.
CC -c -I/opt/pvm3/include Entry.C
CC -o gsm_example +O3 +Oparallel gsm_example.o Entry.o
```

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "Entry.h"
#include <cps.h>

// global data accessible by all threads
char *baseName = 0;
int numEntries = 0;
int numThreads = 0;
barrier_t commonBarrier;
BlockOfEntries **aBlock;

void doWork(void *)
{

     int myIndex = cps_stid(); // get thread id

     // calculate neighbor thread ids
     int rightId = (myIndex!=numThreads-1) ? myIndex+1 : 0;
     int leftId  = (myIndex!=0) ? myIndex-1 : 0;

     char infname[256], outfname[256];
     sprintf(infname, "%s%d", baseName, myIndex);
     sprintf(outfname, "%s_%d", baseName, myIndex);
```

```
    // read in a block of entries
    aBlock[myIndex] = new BlockOfEntries(infname, outfname);

    // synchronize before getting right and left blocks
    cps_barrier(&commonBarrier, &numThreads);

    BlockOfEntries *myBlock = aBlock[myIndex];
    BlockOfEntries *rightBlock = aBlock[rightId];
    BlockOfEntries *leftBlock = aBlock[leftId];

    for(int j=0;j<numEntries/2;j++) {

      // synchronize before updating shadows
      cps_barrier(&commonBarrier, &numThreads);       // update shadow entries
      if (myIndex==0) {
         Entry rightVal = rightBlock->getLeftEnd();
         myBlock->setRightShadow(rightVal);
      }
      else if (myIndex==numThreads-1) {
         Entry leftVal = leftBlock->getRightEnd();
         myBlock->setLeftShadow(leftVal);
      }
      else {
         Entry leftVal = leftBlock->getRightEnd();
         myBlock->setLeftShadow(leftVal);
         Entry rightVal = rightBlock->getLeftEnd();
         myBlock->setRightShadow(rightVal);
      }

      // synchronize after updating shadows
      cps_barrier(&commonBarrier, &numThreads);
      myBlock->singleStepOddEntries();
      myBlock->singleStepEvenEntries();
    }

    delete myBlock; // output sorted entries and delete block

}

main(int argc, char **argv)
{

   if (argc != 4) {
      cout << "Usage: "
           << argv[0]
           << " <base filename> <number of elements> <number of threads>"
           << endl;
      exit(1);
   }   baseName = argv[1];
   numEntries = atoi(argv[2]);
   numThreads = atoi(argv[3]);
```

```
    // allocate space for all the blocks
    aBlock = new BlockOfEntries*[numThreads];

    // set up spawn attributes
    spawn_sym_t spawn_att;
    spawn_att.node = CPS_ANY_NODE;
    spawn_att.min = numThreads;
    spawn_att.max = numThreads;
    spawn_att.threadscope = CPS_THREAD_PARALLEL;

    // create barrier, spawn threads, delete barrier
    cps_barrier_alloc(&commonBarrier);
    cps_ppcall(&spawn_att, doWork, NULL);
    cps_barrier_free(&commonBarrier);

    delete [] aBlock;
}
```

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

S

T

U

V

W

X

Y

Z

B5630-90001