HP 9000 Computer Systems

HP C++ Programmer's Guide

HP 9000 Series Workstations and Servers

HP Part No. 92501-90005
Printed in U.S.A. June 1996

Fourth Edition
E0696

FINAL TRIM SIZE : 7.5 in x 9.0 in

Notice

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, 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. Information in this publication is subject to change without notice.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in sub-paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Rights for non-DoD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19 (c)(1,2).

Printing History

New editions are complete revisions of the manual. Update packages may be issued between editions.

The software code printed alongside the date indicates the version level of the software product at the time the manual was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.

First Edition	March 1990	B1691A.02.00 (Series 300) 92501A.02.00 (Series 800)
Second Edition	December 1990	B2400A.02.10 (Series 300/400) B2404A.02.10 (Series 600/800)
Third Edition	August 1992	B2400A.03.00 (Series 300/400, HP-UX 8.0) B2402A.03.00 (Series 700, HP-UX 8.0) B2404A.03.00 (Series 800, HP-UX 8.0) B2400A.03.05 (Series 300/400, HP-UX 9.0) B2402A.03.05 (Series 700, HP-UX 9.0) B2404A.03.05 (Series 800, HP-UX 9.0)
Fourth Edition	June 1996	HP-UX HP C++ A.03.72 and A.10.22

You may send any suggestions for improvements in this manual to:

Languages Information Engineering Manager
Hewlett-Packard Company
Mailstop 42UD
11000 Wolfe Road
Cupertino CA 95014-9804

Preface

The *HP C++ Programmer's Guide* was written to assist C and C++ programmers execute and debug C++ programs on HP 9000 Series workstations and servers. Although it is not intended as a reference source on the C++ language, you will find a brief overview of the language in Chapter 1. The HP C++ implementation is based on version 3.0 of the C++ translator as developed by USL.

If you are relatively new to C, C++, HP-UX, or the HP Symbolic Debugger, you should become familiar with these languages, systems, and products before using this *Guide*.

This manual contains the following chapters:

Chapter 1 — Overview of HP C++ introduces you to HP C++, providing background information on object-oriented programming and previous releases of C++.

Chapter 2 — The HP C++ Preprocessor presents information about HP C++ preprocessor operation.

Chapter 3 — Compiling and Executing HP C++ Programs describes HP C++ compiler options, system library and header files, and a comprehensive programming example.

Chapter 4 — Optimizing HP C++ Programs describes how your program can be optimized for improved efficiency.

Chapter 5 — Inter-Language Communication describes guidelines for linking HP C++ modules with modules written in HP C, HP Pascal, and HP FORTRAN 77.

Chapter 6 — HP Specific Features of lex and yacc provides a list of HP specific features of the `lex` and `yacc` utilities.

Online Help. The *HP C++ Programmer's Guide* is also available in an online help format. Currently, it is accessible with the `helpprint` command on X and non-X displays, and may also be accessed be selecting the `?` icon on the HP Vue front panel of X displays.

Users with Version A.10.22 or later may also use the command

```
CC +help
```

to access the *HP C++ Online Programmer's Guide*.

The online documentation provides the most comprehensive and current documentation and also provides information on getting help with error messages.

Conventions

NOTATION	DESCRIPTION
`text`	Represents literals; they are to be entered exactly as shown.
italics	Within syntax statements, a word in italics represents a formal parameter or argument that you must replace with an actual value. In the following example, you must replace *filename* with the name of the file you want to compile: `CC` *filename*
punctuation	Within syntax statements, punctuation characters (other than brackets, braces, vertical parallel lines, and ellipses) must be entered exactly as shown.
{ }	Within syntax statements, braces indicate that you must choose one of the listed items. In the following example, you must specify either `ON` or `OFF`: `#pragma OPTIMIZE { ON | OFF }`
[ ]	Within syntax statements, brackets enclose optional elements. In the following example, brackets around *optionarg* indicate that the argument is optional: *-optionname[optionarg]*
[|]	A vertical bar within brackets indicates that you can choose either or both of the items separated by the vertical bar. In the following example, you can specify either *options* or *files* or both: `CC [` *options* `|` *files* `]`
[. . .]	Within syntax statements, a horizontal ellipsis enclosed in brackets indicates that you can repeatedly select elements that appear within the immediately preceding pair of brackets or braces. In the following example, you can select *itemname* and its delimiter zero or more times. Each instance of *itemname* must be preceded by a comma:

`[,`*itemname*`[...] ]`

If a punctuation character precedes the ellipsis, you must use that character as a delimiter to separate repeated elements. However, if you select only one element, the delimiter is not required. In the following example, the comma cannot precede the first instance of *itemname*:

`[`*itemname*`][,...]`

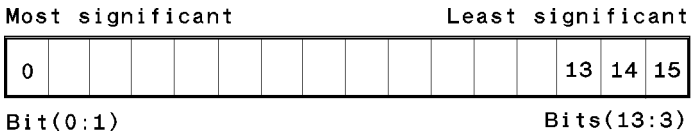
... ⋮ Within examples, horizontal or vertical ellipses indicate where portions of the example are omitted.

base prefixes The prefixes %, #, and $ specify the numerical base of the value that follows:

%num specifies an octal number.
#num specifies a decimal number.
$num specifies a hexadecimal number.

When no base is specified, decimal is assumed.

Bit (*bit:length*) When a parameter contains more than one piece of data within its bit field, the different data fields are described in the format Bit (*bit:length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, Bits (13:3) indicates bits 13, 14, and 15:



Contents

Figures

Tables

1

Overview of HP C++

C++ is rapidly emerging as a popular successor to C. The C++ language retains the advantages of C for systems programming, while adding features and extensions that make it easier and safer to use. Moreover, C++ supports object-oriented programming. You can use object-oriented programming techniques to write applications that are typically easier to maintain and extend than non-object-oriented applications.

This manual describes HP C++, which is Hewlett-Packard's implementation of the C++ programming language for systems running HP-UX. HP C++ is derived from version 3.0 of the USL product.

C++ translator, which translates C++ source code into C code. However, with HP C++ you can *compile* C++ source code *directly to object code*, as well as translate C++ code into C code.

HP C++ is a compiling system that enables you to develop executable files from C++ source code. The components of the compiling system are driven by the `CC` command line interface. The various components preprocess and compile the C++ source files, add information needed for debugging, and link the resulting object files.

This chapter

- provides a brief history of C++
- tells you the difference between C and C++
- explains how to compile and execute a simple C++ program
- describes object-oriented programming with C++
- highlights the incompatibilities and differences between HP C++ and previous releases of C++

History of C++

C++ is a general-purpose programming language designed at AT&T Bell Laboratories and licensed through USL.

Based on the C programming language, C++ was designed to be used in a C programming environment on a UNIX system. C++ retains most of C's efficiency and flexibility, incorporates all the features of C, and also supports features that are unavailable in the C language. Many of the added features were designed to support object-oriented programming.

Dr. Bjarne Stroustrup, author of *The C++ Programming Language*, designed most of the new language, with additional contributions from Brian Kernighan and other Bell Labs staff. In undertaking the project, Stroustrup borrowed successful features from other older languages. As a result, C++ incorporates the concepts of classes and virtual functions from Simula67. C++ borrows the idea of operator overloading from Algol 68. These features are an important part of the support that C++ provides for object-oriented programming.

Early versions of the language were collectively known as "C with Classes" and lacked many details that were added later. According to Stroustrup, the name C++ was coined by Rick Mascitti. The name is a play on words since "++" is the C increment operator and can also be taken to signify the evolution of changes from C. Stroustrup also points out that the language is not called D because it does not remove any features of C, but rather it is an extension of C.

The USL translator has evolved through several releases. Version 1.0, the original release, reflects the language as defined in Bjarne Stroustrup's *The C++ Programming Language*. Version 1.1 added two features: pointers to member functions and the keyword `protected`. Version 1.2 added support for the overloading of unsigned integers and unsigned longs.

Version 2.0 added several major features, including support for multiple inheritance, additional operator overloading, and type-safe linkage. Version 2.0 also fixed a number of problems in the C++ language. As a result, version 2.0 is not backward compatible with previous releases.

Version 2.1 primarily repaired defects and more rigorously enforced the definition of the language. In addition, HP C++ added compiler mode to version 2.1, which compiles C++ source *directly to object code* instead of

translating it to C. This reduces compilation time significantly. Version 2.1 is both source compatible and link compatible with version 2.0.

The C++ Programming Language, written by Bjarne Stroustrup, contains the definition of the C++ language supported by the current version, 3.0. (Language features that are not implemented in version 3.0 are listed in appendix C, "Not Implemented Messages," of the *C++ Language System Release Notes*.) Version 3.0 adds significant new functionality in templates, true nested classes, protected derivation, and a number of other new features.

HP C++ implements version

3.0 of the USL translator and adds an exception handling mechanism that conforms to the definition in *The C++ Programming Language*.

HP C++ also supports shared libraries on HP-UX by allowing you to create position-independent code (PIC).

Getting Started with HP C++

This section briefly describes the use of the `CC` command to invoke HP C++, tells you how to compile and execute a simple C++ program, and explains how to access online sample programs.

Using the CC Command

To invoke the HP C++ compiling system, use the `CC` (uppercase) command at the shell prompt. The `CC` command invokes a driver program that runs the compiling system according to the filenames and command line options that you specify. See Chapter 3 for more details about the compiling system and the `CC` command.

Compiling and Executing a Simple Program

The best way to get started with HP C++ is to write, compile, and execute a simple program. The following is a simple program to get you started:

```
#include <iostream.h>
void main()
{
      int x,y;
      cout << "Enter an integer: ";
      cin >> x;
      y = x * 2;
      cout << "\n" << y <<" is twice " << x <<".\n";
}
```

Compiling this example with `CC` produces an executable file named `a.out`. To run this executable file, just enter the name of the file. The following summarizes this process with the file named `getting_started.C`:

```
$ CC getting_started.C
$ a.out
Enter an integer: 7

14 is twice 7.
```

Debugging C++ Programs

You can debug your C++ programs with the HP Symbolic Debugger. You need to compile your program with the `-g` option first. For more information about the HP Symbolic Debugger, see the *HP-UX Symbolic Debugger User's Guide*.

Using the Online Sample Programs

Many of the C++ programs from this and other manuals are stored online for you to use and experiment with. The source files for these programs reside in the directory `/usr/contrib/CC/Examples` (`/opt/CC/contrib/Examples/bank_ex` for HP-UX 10.*x* C++ versions).

How C++ Differs from C

C++ is often described as a superset of C because C++ has many of the features of C, plus some additional features. There are, however, some differences between the two languages aside from the additional features of C++. This section briefly describes the following:

- Compatibility with C
- Reliability Improvements
- Other Extensions to C
- Changing Your C Programs to C++

C++ also differs from C in its support of object-oriented programming. Refer to "Support for Object-Oriented Programming" for a discussion of C++ as an object-oriented programming language. For more details about the C++ language, refer to the *The C++ Programming Language*.

Compatibility with C

Retaining compatibility with C served as a major design criterion for C++. The basic syntax and semantics of the two languages are the same. If you are familiar with C, you can program in C++ immediately.

For instance, C++ preserves C's efficient interface to computer hardware. That is, C++ has the same types, operators, and other facilities defined in C that usually correspond directly to computer architecture. You can use these facilities to write code that makes optimal use of the hardware at run time (for example, code that manipulates bits and uses register variables).

C++ also preserves and enhances the C facilities for designing interfaces among program modules. These facilities are extremely useful when you develop any size application, but particularly a large or complex one.

Finally, C++ modules can usually be linked with already existing C modules with few if any modifications to the C files. This means that you can probably use many C libraries with your C++ programs.

Refer to "Changing Your C Programs to C++," in this chapter, for a description of specific things you might want to change in order to convert

existing C programs to C++ programs. Refer to Chapter 5, "Inter-Language Communication," for details about linking C programs with C++ programs.

Reliability Improvements

C++ provides several features to help you create more reliable programs. These features include type checking, constant data types, and flexibly located variable declarations. The following sections briefly describe these features.

Type Checking Features in Functions

You declare functions in C++ just as you do in C, except that C++ supports type checking of arguments. Type checking means that the compiling system detects many errors at compile time rather than at run time, so you can correct them earlier in the development process.

Unlike pre-ANSI C functions, C++ functions must specify types for function arguments. Furthermore, the compiling system performs type checking and type conversion. This means that it compares the argument types with the parameter types in a function definition each time the function is called. If the types are not compatible, the compiling system generates an error. For example, suppose you define a function `max` and then make the function calls shown in the following code fragment:

```
float max(float x,float y)       // Define a function, max.
{  return (x>y) ? x : y;  }
    ⋮
max (4.0, 9.0);      // This function call will compile since
                     // both arguments are floats.
max(4,9);            // This function call will compile since
                     // the function arguments are integers,
                     // which can be converted to floats.
max("Four",9.0);     // WRONG!
                     // First argument is a character string, which
                     // is an incorrect type, and conversion is not
                     // possible.
```

C++ provides function argument checking that is compatible with ANSI C. C++ also provides type-safe linkage with checking done at run time, unlike C.

Constant Data Types

C++ provides a new keyword, `const`, that declares an identifier to be a constant. A similar feature is also part of ANSI C. For example, the following line creates a variable `days`, which behaves exactly like any other `int` variable except that its value cannot be changed:

```
const int days = 7;   // Days is an integer constant.
```

You can also use `const` with pointers, either to declare an object pointed to as a constant or to declare the pointer itself as a constant.

You can use a `const` declaration in a C++ program in many places where you would have used a `#define` macro in a C program. Unlike constants created by `#define` macros, which are purely textual substitutions, `const` values can have type and scope like variables.

Variable Declarations

C++ allows you to declare variables and statements in any order, as long as you declare variables before you use them. You can declare variables almost anywhere in a block, not just at the beginning. As a result, you can locate variables with the statements that use them. For example, the following fragment is legal in C++:

```
for (int j = 0; j < 100; j++)
```

The example shows how C++ allows you to declare and initialize the variable `j` in the `for` loop statement instead of at the start of the function.

Other Extensions to C

The previous sections describe how C++ can improve reliability. Other features of C++ distinguish it from C. Many of these additional features add to its support for object-oriented programming as well as to its stronger type checking. This section describes these additional features very briefly. Refer to "Support for Object-Oriented Programming" for details about object-oriented programming.

Comments

C++ has the following two notations for comments:

- Comments can begin with the characters `/*` and end with `*/`, as they do in C.
- Any line that begins with // is a comment, and any text following // until the end of the line is a comment.

You can use both styles of notation in the same file.

For example,

```
/* This is a C-style comment that extends
over more than one line; it is also a
legal comment in C++ */

// This is a C++-only comment that
// extends over more than one line
```

Here's another example of a C++ comment:

```
int i; // counter variable
```

Default Arguments

To account for missing arguments in a function call, function declarations and definitions can specify default expressions for the arguments. You declare these default expressions simply by initializing the arguments. The initialized values are called default values. For instance, the following code fragment shows two default arguments:

```
// default values are 0 and "none"
void add_items (int i=0, char *str = "none");
```

When a call to `add_items` is missing an argument, the default value is substituted in its place. If a call to `add_items` has two arguments, then the default values are ignored. You can provide default values for trailing arguments only. Trailing arguments are the last arguments in the argument list.

Variable Number of Arguments

In addition to specifying argument types, a C++ function declaration can specify that a variable number of arguments is accepted. This is also a feature of ANSI C.

You declare a function with variable arguments by adding ellipsis points (...) to the end of the declaration of the function's argument list. The ellipsis instructs the compiler to accept any number of actual arguments of any type from that point on in the argument list of a function call. For example, the following function can be called with a variable number of arguments:

```
int file_print(FILE*, char*, ...);
```

The preceding code fragment declares that `file_print` is a function that returns an integer and can be called with a variable number of arguments, the first two of which must be of the types `FILE*` and `char*`, respectively.

Overloaded Functions

C++ supports *function name overloading*, which allows you to give the same name to different functions. You typically use function overloading for functions that perform the same operations on objects of different types. The compiling system determines which function to use based on the number and type of arguments that are passed.

For example, you might want to define two functions named `print`. One can be used for printing integers and the other for printing character strings. Or you might want to be able to handle information about dates as integers (when you want to do calculations) or as characters (when you want to display them). The following code fragment illustrates a function, `handle_date`, that handles dates as either integers or characters.

```
// This function takes three arguments that must be integers.
void handle_date(int day, int month, int year);

// This function takes one argument that must be a pointer
// to a character.
void handle_date(char* date);
```

Note Releases of the translator before version 2.0 required the use of the keyword `overload` to specify that a name could be used for more than one function. As of version 2.1, the keyword `overload` is obsolete.

Changing Your C Programs to C++

This section contains information about changes you might want to incorporate into C programs and header files that you intend to use with HP C++. These changes are in the following categories:

- new keywords
- function declarations
- structures
- external names
- constants
- assignment of void pointers

When you start to use C++ after using C, you might also want to change to an object-oriented approach to programming. Refer to "Support for Object-Oriented Programming" for details about object-oriented programming with C++.

New Keywords

C++ reserves as keywords the following identifiers that are not keywords in HP C:

C++ Keywords

`catch`	`new`	`this`
`class`	`operator`	`throw`
`const`[1]	`private`	`try`
`delete`	`protected`	`virtual`
`friend`	`public`	`volatile`[1]
`inline`	`template`	

1 The keywords `const` and `volatile` are also keywords in ANSI C.

If your C code contains any variables with these names, you must change them when you convert your program to a C++ program.

Note Although it is reserved as a keyword, `volatile` is not implemented in HP C++. However, the `+Ovolatile` optimization option makes all global variables `volatile`, and performs level 2 optimization.

Function Declarations

You should make the following changes involving functions:

- Explicitly declare all functions. (You cannot use implicit declarations in C++.)
- Add argument types to function declarations.
- Use ellipsis points (...) for functions that take varying numbers of arguments.

One important difference between C and C++ is that a C++ function declared as `f()` takes no arguments, whereas a C function declared as `f()` can take any number of arguments of any type. This means that you do not need to use

`void` to declare that a C++ function takes no arguments, as you might have done in an ANSI C program.

Unlike ANSI C, C++ does not require a comma separating the ellipsis points from the rest of the argument list when you specify a variable number of arguments.

Structures

Since a C++ `struct` is a particular form of the `class` data type, you may need to change your C code to avoid name conflicts.

External Names

In C you can define a variable in an external file more than once. The last initializer read by the linker is the variable's initial value at run time. In C++ you can define a variable declared in an external file exactly once. For example, the following code is legal in C but not in C++:

```
/* this is a C program but not a C++ program */

#include "file1.c"
#include "file2.c"
extern int foo();
main()
{
    ⋮
            }

/* file1.c */
int i ;          /* i is also defined in file2 */
int foo () { return i; }

/* file2.c */
int i;           /* i is also defined in file1 */
int fum() { return foo(); }
```

If you try to compile this program with `CC`, you get the following error message:

```
CC: "file2.c", line 2: error: two definitions of ::i (1034)
```

In this example, you can eliminate the error generated by `CC` by specifying the second definition of `int i` to be `extern`, as follows:

```
/* file2 */
extern int i;    /* i is also defined in file1 */
int fum() { return foo(); }
```

Constants

ANSI C constants are stored as `extern`, whereas C++ constants are, by default, `static`, although they can be declared `extern`. In other words, if you define a file scope `const` in ANSI C with no storage class (that is, neither `static` nor `extern`), the linkage defaults to `extern`. This is an important difference between types in ANSI C and C++. Hence, the following compiles and links using ANSI C:

```
/* fileA.c */
const int x = 100;

/* fileB.c */
#include <stdio.h>

main()
{
     extern const int x;
     printf("%d\n", x);
}
```

These files also compile using HP C++, but fail to link, with the following error:

```
/bin/ld: unsatisfied symbols
     x (data)
```

The constant `x` defined in `fileA.c` has no "linkage." To make `x` externally visible, you must explicitly give it the storage class `extern`, as shown below:

```
extern const int x = 100;
```

Assignment of Void Pointers

C++ does not allow you to assign a void pointer to another pointer variable. For instance, the following code is legal in C, but illegal in C++:

```
char* cp;
void* vp;
cp = vp;          // WRONG!
```

You must use a cast, as shown below:

```
cp = (char *) vp;
```

Character Array Initialization

Character array initialization is handled differently in C++ and ANSI C. For more information refer to *The C++ Programming Language*.

Support for Object-Oriented Programming

C++ supports object-oriented programming; C does not. This section describes object-oriented programming, gives a brief example of an object-oriented approach to a programming problem, and gives an overview of the language enhancements that C++ provides for object-oriented programming.

What Is Object-Oriented Programming?

The traditional approach to programming is often summarized by the equation:

PROGRAM = DATA STRUCTURES + ALGORITHMS

According to this approach, a program is a blend of data (information) and algorithms (procedures). The data is the information given in a problem that may be useful in obtaining a solution. Procedures are the steps you take in manipulating the data to obtain a solution to the problem. Procedural programming, or non-object-oriented programming, typically focuses initially on the procedures. The key to a clever procedural program is often a clever algorithm.

Object-oriented programming shifts the emphasis from algorithms, or *how* things get done, to object declarations, or *what* needs to be manipulated. The object-oriented programmer typically starts by developing a concept of an object or collection of objects whose state and functionality are independent of a particular program.

Moreover, in an object-oriented program, the concept of procedure and data is replaced by the concept of objects and messages. An *object* is a package containing two components: information and a description of how to manipulate the information. A *message* specifies one of an object's manipulations. To send a message is to tell an object what do. The object determines exactly what methods to use. For example, a message to a `circle` object in an object-oriented graphics program might say "draw yourself."

In other words, object-oriented programming rejects the dichotomy between data and procedures and substitutes the concepts of objects (which contain both data and procedures) and messages:

PROGRAM = OBJECTS + MESSAGES

Object-Oriented Programming: The Bank Example

For example, suppose you want to develop a program that a bank can use to keep track of its transactions. Most of its transactions have to do with money and customers. Customers can borrow, save, invest, or write checks on their money, and most of the bank's money is kept in accounts.

A programmer using a non-object-oriented approach might develop a solution to the bank's needs by analyzing the bank's various transactions and turning these transactions into program routines. For example, there might be routines with names such as `calculate_interest` and `add_deposit` that pass and return arguments containing data about money, customers, and accounts.

A programmer using an object-oriented approach, in contrast, would probably begin by thinking of the objects in the bank rather than the bank's transactions. An object-oriented language would allow an object such as an `account` or a `customer` to contain both the information needed to define the object *and* the functions that define operations that can manipulate the object. Thus, an `account` object might contain an amount of money and also a function to calculate and add interest to its amount of money.

In the banking example, this concept of an `account` object allows you to send a message to an `account` object telling the object to update its balance. Upon receiving this message, the `account` object manipulates its data according to its own definitions of how to carry out the operations requested in the message.

Furthermore, the programmer using an object-oriented approach might design the bank program to include a hierarchy of `account` objects. All `account` objects could be derived from `account` and therefore contain whatever data and operations are part of an `account`. Moreover, the derived objects might also have additional or slightly different data or operations. Thus, a `checking_account` might contain a function that sets the interest for the account at a rate lower than the interest for a `savings_account`.

The `bank_example` program in example 1-1 is intended to illustrate these object-oriented programming concepts. It is not intended to represent a realistic application. The next several sections refer to the `bank_example` program. The source file for this program resides in the directory `/usr/contrib/CC/Examples` (`/opt/CC/contrib/Examples` for HP-UX 10.*x* C++ versions).

```
//*********************************************************
//program name is "bank_example"
//*********************************************************
#include <iostream.h>  // needed for C++-style I/O
#include <string.h>    // needed for C-style string manipulation
class account
{
private:
    char* name;         // owner of the account
protected:
    double balance;     // amount of money in the account
    double rate;        // rate of interest for the account
public:
    account(char* c)    // constructor
       {
       name = new char [strlen(c) +1]; strcpy(name,c);
       balance = rate = 0;
       }
    ~account()          // destructor
       { delete name; }
              // add an amount to the balance
    void deposit(double amount) { balance += amount; }
              // subtract an amount from the balance
    void withdraw (double amount) { balance -= amount; }
              // show owner's name and balance
    void display()
       { cout << name << "   " << balance << "\n"; }
              // this function is redefined for
              // checking_account, which is a derived class
    virtual void update_balance()
       { balance += (  rate * balance ); }
};
```

Example 1-1. Object-Oriented Programming with C++: bank_example

```
                // define a class derived from account
class checking_account : public account
{
private:
    double fee;  // checking accounts have a fee in
                 // addition to name, balance, and rate
public:
         // constructor; note that checking accounts
         // pay 5% interest but charge $2.00 fee
    checking_account(char* name) : account(name)
       { rate = .05; fee = 2.00; }
         // redefined to deduct fee for this
         // type of account
    void update_balance()
       { balance += (  rate * balance ) - fee; }
};

         // define a class derived from account
class savings_account : public account
{
public:
         // constructor; note that savings accounts
         // pay 10% interest and charge no fee
    savings_account(char* name) : account(name)
       { rate = .10; }
};

main()
{
  checking_account* my_checking_acct =
      new checking_account ("checking");
  savings_account* my_savings_acct =
      new savings_account ("savings");
         // send a message to my_checking_acct
         // to display itself
```

Example 1-1. Object-Oriented Programming with C++: bank_example (continued)

```
  my_checking_acct->display();
         // send a message to my_savings_acct to
         // display itself
  my_savings_acct->display();
         // send a message to my_checking_acct
         // to deposit $100 to itself
  my_checking_acct->deposit(100);
         // send a message to my_savings_acct
         // to deposit $1000 to itself
  my_savings_acct->deposit(1000);
         // send a message to my_checking_acct
         // to update its balance
  my_checking_acct->update_balance();
         // send a message to my_savings_acc
         // to update its balance
  my_savings_acct->update_balance();
         // send a message to my_checking_acct
         // to display itself
  my_checking_acct->display();
         // send a message to my_savings_acct
         // to display itself
  my_savings_acct->display();
}
//***********************************************
```

Example 1-1. Object-Oriented Programming with C++: bank_example (continued)

When you compile and run the `bank_example` program, you get the following results:

```
checking     0
savings      0
checking     103
savings      1100
```

How Does C++ Support Object-Oriented Programming?

To support object-oriented programming, a language must support the following:

- *Encapsulation* — All the functions that can access an object are in one place and data and functions can be defined that can only be accessed from within that specific class.
- *Data abstraction* — You can define data types that can be used without knowledge of how they are represented in storage.
- *Inheritance* — You can develop hierarchies of objects that inherit data and functionality from their parent objects.
- *Type polymorphism* — A pointer to an object can point to a variety of different types, and you can use a process called dynamic binding to select and execute an appropriate function at run time based on the type of the object that is actually referenced.

C++ has all of these characteristics, which are described in more detail in the following sections.

Encapsulation

Encapsulation means that all the functions that can access an object are in one place. C++ supports the `class` data type, which allows you to declare all the functions that can access its data within the body of its declaration. A `class` is a lot like a structure in C and it is the basis for much of the support that C++ provides for object-oriented programming.

For example, suppose you are using C++ to develop the banking application described briefly in the preceding section. You could define a `class` to represent an `account` object. Its *data members* could represent the customer who owns the account, the balance of the account, and the rate of interest for the account. Its *member functions* could specify operations to be used with the data members. Your code, nearly identical to that in example 1-1, might look something like the following fragment:

```
class account
{
  private:
     char* name;         // owner of the account
     double balance;     // amount of money in the account
     double rate;        // rate of interest for the account
  public:                // add an amount to the balance
     void deposit(double amount) { balance += amount; }
                         // subtract an amount from the balance
     void withdraw (double amount) { balance -= amount; }
                         // show owner's name and balance
     void display() { cout << name << "   " << balance << "\n"; }
                         // add interest to the balance
     void update_balance() { balance += (  rate * balance ); }
};
```

In this example, `account` is a class. The keywords `private` and `public` divide the class into two parts. The members in the first part — the `private` part — are data members. The members in the second part — the `public` part — are member functions. Because they are defined to be `private`, the data members, specifically `name`, `balance`, and `rate`, can only be used by member functions of the `account` class. In other words, the only functions that can access `name`, `balance`, and `rate` are `deposit`, `withdraw`, `display`, and `update_balance`.

Figure 1-1 illustrates this definition of an `account` class. The arrow in the figure indicates that the functions in the `public` part of the `account` class have access to the data in the `private` part of the class.

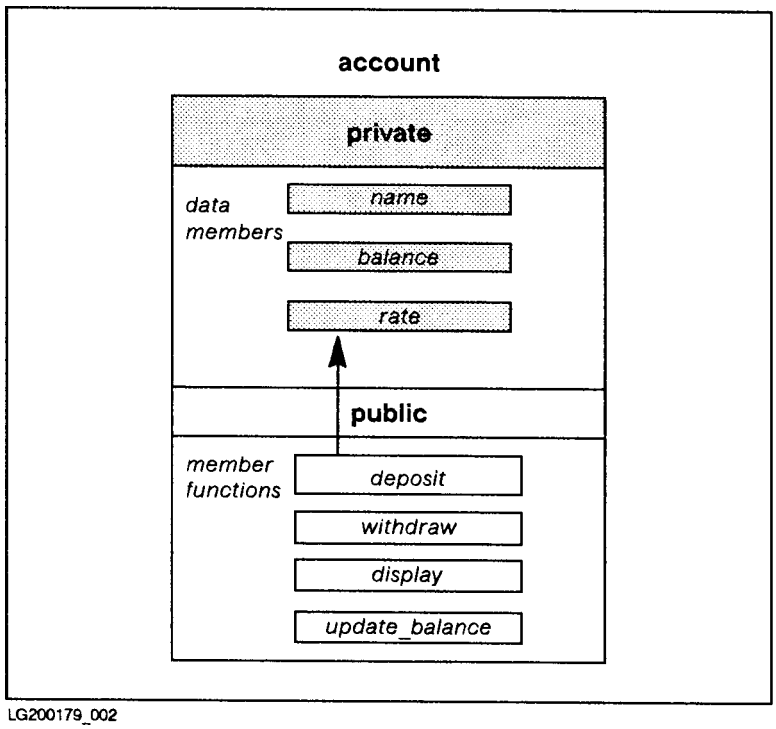


Figure 1-1.
Encapsulation in a C++ Class: The account class Example

Note that some or all of the data members could have been `public` and some or all of the member functions could have been `private` or `protected` in the `account` class. Refer to "Inheritance" for more information on the keyword `protected`. The design shown in Figure 1-1 is only one of many ways to use encapsulation in defining classes.

Data Abstraction

C++ classes allow you to hide the representation of data in storage as well as restrict access to data. In other words, classes allow you to define data types that can be used without knowledge of their representation in storage.

You can use C++ classes in the same way that you use built-in types. For example, `float` is a built-in type. To use a `float` object you do not need to know how the object is represented in storage. All you need to know is the name of the type and the operations that you are allowed to perform on that type. When you use floating-point objects, you can add or assign values to them without concern for their representation. The representation of the objects is hidden.

Similarly, C++ lets you use a class like `account` while ignoring the details of how an `account` object is represented. All you need to know is that accounts have owners, balances, and interest rates, that you can make deposits and withdrawals, that you can display the name of the account's owner and balance, and that you can update the balance.

Furthermore, you can use data abstraction to design large or complex applications with many pieces that use objects of a class in different ways. If you need to change the representation of a class, you only need to do so in one place. Also, you can add modules that use objects of the class in entirely new ways.

Finally, data abstraction means that access to the representation of data objects is restricted. Restricting access to data makes debugging easier and assists you in protecting the integrity of class objects. For instance, C++ allows you to trace an error involving the private members of a class to the limited number of functions that have access to that data. Thus, an error involving the `name` data in the private part of an `account` object probably arises from one of the `account` member functions (`deposit`, `withdraw`, `display`, and `update_balance`), since they are the only functions allowed to access `name` data. Similarly, the representation of `name` data is consistent for all applications using `account` objects, since it is only accessible to `account` member functions.

Inheritance

C++ supports inheritance, allowing you to derive a class from one or more base classes. For example, using the class `account` as a base class, you can define derived classes named `checking_account` and `savings_account` as shown in the following fragment taken from Example 1-1:

```
                // define a class derived from account
class checking_account : public account
{  . . .
};
                // define a class derived from account
class savings_account : public account
{  . . .
};
```

Figure 1-2 illustrates the concept of single inheritance: checking and savings accounts are each derived from a base class.

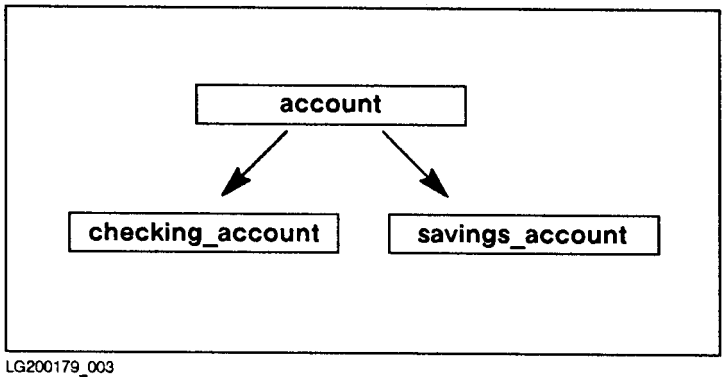


Figure 1-2. Concept of Single Inheritance: The account Example

Multiple inheritance means that a class can have more than one base class. For instance, you could define a `savings_account` object as derived from an `investment` object as well as from `account`. Other objects derived from `investment` might represent stocks and real estate. This concept of multiple inheritance is shown in Figure 1-3.

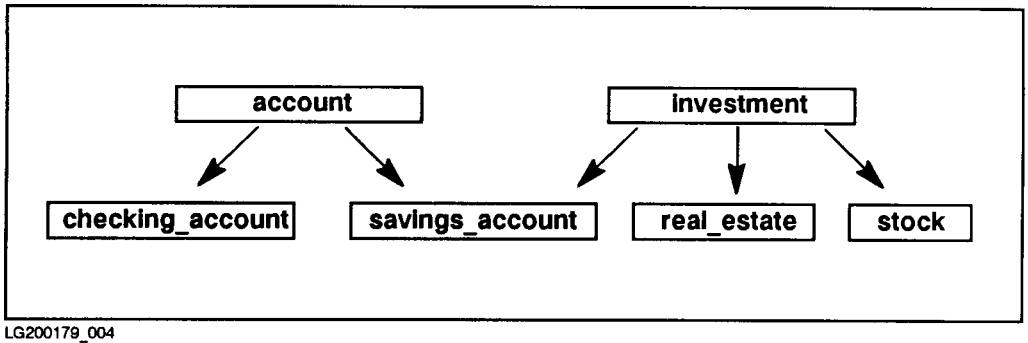


Figure 1-3. Concept of Multiple Inheritance: The savings_account Example

Deriving classes allows you to define details common to many potential derived classes in a base class. Derived classes inherit all members—both data and functions—of the base class. Thus all `checking_account` and `savings_account` objects inherit `name`, `balance`, and `rate` data members from the base class `account`, as well as the public member functions `deposit`, `withdraw`, `display`, and `update_balance`. In order for the derived class to access inherited members, however, the members must be declared `public` or `protected`. The keyword `protected` allows the derived class to access a member of the base class, while blocking access to the rest of the program.

Inheritance allows you to write the source code for a base class, store the declarations in a header file, and then use the base class to derive new classes with additional data or functions. This means that you can write separate modules that extend a large application without affecting the header file. For instance, in the modified `bank_example` program, in addition to inheriting `name`, `balance`, and `rate` data from the base class, `checking_account` objects have a new data member, `fee`.

Furthermore, C++ allows you to redefine a base class's member functions in each class derived from it. For instance, suppose in the `bank_example` program you had defined the `update_balance()` function in `account` as follows:

```
void update_balance() { balance += ( rate * balance ); }
```

Since checking accounts charge fees, the `checking_account` version of `update_balance()` was redefined to deduct a fee as well as add interest as follows:

```
void update_balance() { balance += ( rate * balance ) - fee; }
```

Type Polymorphism

As was mentioned previously, *type polymorphism* means that a pointer to an object can point to a variety of different types, and you can select and execute an appropriate function at run time based on the type of the object actually referenced. The rest of this section discusses how C++ implements the concept of type polymorphism using dynamic binding, inheritance, and type checking.

In C++, a pointer to a derived class is type-compatible with a pointer to its base class. As a result, it is possible for a pointer declared as the address of one class type to be assigned the address of another type.

For instance, as was mentioned previously, if `checking_account` and `savings_account` are both derived from `account`, the following is legal:

```
    // account_ptr points to an account object
account* account_ptr;
    // checking_ptr points to a checking_account object
checking_account* checking_ptr;
    // savings_ptr points to a savings_account object
savings_account* savings_ptr;
    // now account_ptr points to a savings_account object
account_ptr = savings_ptr;
    // now account_ptr points to a checking_account object
account_ptr = checking_ptr;
```

In other words, a variable declared as a pointer to a particular class might actually point to an object of a different class at run time. In this example, `account_ptr` points to a `checking_account` rather than an `account`.

While this type compatibility can be convenient, it can also result in ambiguity as to which class member function should be called. For instance, after making the preceding declarations and assignments, suppose you make the following function call:

```
    // Does this call the account member function
    // or the checking_account member function?
account_ptr->update_balance();
```

C++ handles this ambiguity by allowing you to specify a base class member function as `virtual`. When you declare a function to be virtual, you tell the compiling system to select and execute the appropriate function at run time depending on an object's actual type, rather than its declared type. This is called *dynamic binding*.

For example, consider the `update_balance` function, which is defined as `virtual` in the following code fragment taken from `bank_example`:

```
class account
{  . . .
         // this function is redefined for checking_account,
         // which is a derived class
    virtual void update_balance()
       { balance += ( rate * balance ); }
};
         // define a class derived from account
class checking_account : public account
{  . . .
    void update_balance()
       { balance +=  ( rate * balance ) - fee; }  };
```

Declaring `update_balance` as `virtual` means that the compiling system uses dynamic binding. Here's how it works in the `bank_example`:

- Suppose you declare `account_ptr` as a pointer to an `account` object, and you assign it the address of a `checking_account` object. Then you make a function call: `account_ptr->update_balance()`.
- C++ uses the `checking_account` definition of `update_balance`. This means that the fee is deducted from the account balance.
- If you do *not* declare `update_balance` as `virtual`, C++ would use the `account` version of `update_balance`, ignoring the fact that `account_ptr` actually points to a `checking_account` object.

Inline Functions

Calling small functions frequently can slow a program's execution speed. Therefore, C++ allows you to declare *inline expanded* functions using the keyword `inline`. This means that the compiling system attempts to generate the code for the function at the place where it is called. When used with small functions, `inline` can increase execution speed.

If a function is not a class member, you can make it inline by declaring it with the `inline` specifier as shown in the following example:

```
inline int max (int a, int b) {return a > b ? a : b;}
```

Inline expansion is especially useful for defining small member functions. A member function becomes inline when it is defined within the definition of its class. For example, the `show_radius` function in the following code is inline expanded whenever possible:

```
class circle                // declare a class
{
    double radius;
public:
    void show_radius()
       { cout << "radius is " << radius ;}  // an inline function
};
```

The new and delete Operators

You can declare a named object in C++ to be `static` or `auto` just as you can in C. A *static object* is created once at the start of the program and destroyed once at the termination of the program. Its scope is the block in which it is declared; if it is declared outside of a block, it has file scope. An *automatic object* is created each time its declaration is encountered in the execution of the program and destroyed each time the block in which it occurs is exited. Its scope is from the point of declaration to the end of the block in which it is declared.

You can also control the life span of an object and allocate storage just for the time the object is needed. The operator `new` creates objects and the operator `delete` destroys them. Using these operators, `new` and `delete`, allows you to

use an object created by a function after leaving the function. The objects created by `new` are allocated from free storage.

Because they are built into the language, `new` and `delete` are easier to use than `malloc` and `free`, which are not built into the C language. (The functions `malloc` and `free` are UNIX library calls.) C++ also allows you to overload `new` and `delete`, which means that you can create your own memory management operators on a class by class basis. Moreover, the `new` and `delete` operators for class objects invoke constructors and destructors, which are described briefly in the next section.

Constructors and Destructors

Constructors guarantee initialization of class objects; they are member functions designed explicitly to initialize objects. A constructor sets up and assigns a value in storage when a class object is declared.

Many classes also have *destructors*. A destructor ensures that storage is released, that counters are reset, and that other maintenance takes place when class objects are destroyed (for example, when a variable goes out of scope).

A constructor has the same name as its class, whereas a destructor for a class is the class name preceded by a tilde (~). Thus for class `account`, a constructor is named `account` and its destructor is named `~account`. These are shown in the following code fragment:

```
account(char* c)     //constructor
      {
      name = new char [strlen(c) +1]; strcpy(name,c);
      balance = rate = 0;
      }
~account()           // destructor
       { delete name; }
```

Note that destructors can also be declared as `virtual` functions, thus ensuring that the appropriate destructor is always called regardless of its apparent type. (Refer to "Type Polymorphism" above.)

Overloaded Operators

Classes can have functions that assign special user-defined meanings to most of the standard C++ operators when they are applied to class objects. These functions are called *overloaded operator* functions. For example, if you are designing an application using complex numbers, you could overload the plus (+) operator to handle complex addition. The following code fragment illustrates such an application:

```
class complex
{
         // the real and imaginary parts of the number
  double real, imag;
public:
  complex(double r,double i) // constructor
     { real = r; imag = i; }
         // declare overloaded "+" operator
         // as a member function
  complex operator+(complex addend);
};
```

The name of an operator function is the keyword `operator` followed by the operator itself, such as `operator+`. You can declare and call an operator function in the same way you call any other function, by using its full name, or by using just the operator. When you use just the operator, C++ selects the correct overloaded operator function to perform the task. An operator function can be a member function.

Conversion Operators

Conversion operators are member functions that have the same name as a type. The type can be either user-defined or built-in. You can use conversion operators to define your own type conversions. Declare a conversion operator as an overloaded operator function with the keyword `operator`.

For example, the following code fragment defines a conversion operator for a conversion from `circle` (a user-defined type) to `int` (a built-in type):

```
class circle
{
private:
  int radius;
public:
    :
  operator int()   // conversion operator defines a
                   // conversion from a circle to integer
         { return radius; }
};
```

Given the preceding definition, you could make the following declaration and assignment:

```
circle A(1);  // create a circle A with a radius of 1
int i = A;    // initialize an integer variable, i,
              // by converting A to an int and
              // assigning the result to i
```

Templates

Version 3.0 of HP C++ adds templates, or parameterized types as they are also called. This section briefly describes templates. For a detailed description, see the "Template Instantiation" sections in the *C++ Language System Selected Readings* and the "Templates" chapter in the *The C++ Programming Language*. For additional information on templates, see the *HP C++ Online Programmer's Guide*.

You can create class templates and function templates. A template defines a group of classes or functions. The template has one or more types as parameters. To use a template you provide the particular types or constant expressions as actual parameters thereby automatically creating a particular object or function.

Class Templates

A class template defines a family of classes. To declare a class template, you use the keyword `template` followed by the template's formal parameters. Class templates can take parameters that are either types or expressions. You define a template class in terms of those parameters. For example, the following is a class template for a simple stack class. The template has two parameters, the type specifier `T` and the `int` parameter `size`. The keyword `class` in the `< >` brackets is required to declare a template's type parameter. The first parameter `T` is used for the stack element type. The second parameter is used for the maximum size of the stack.

```
template<class T, int size>
class Stack
{
public:
     Stack(){top=-1;}
     void push(const T& item){thestack[++top]=item;}
     T& pop(){return thestack[top--];}
private:
     T thestack[size];
     int top;
};
```

The member functions and the member data use the formal parameter type `T` and the formal parameter `size`. When you declare an instance of the class `Stack`, you provide an actual type and a constant expression. The object created uses that type and value in place of `T` and `size`, respectively. For example, the following program uses the `Stack` class template to create a stack of 20 integers by providing the type `int` and the value `20` in the object declaration:

```
void main()
{       Stack<int,20> myintstack;
        int i;

        myintstack.push(5);
        myintstack.push(56);
        myintstack.push(980);
        myintstack.push(1234);
        i = myintstack.pop();
}
```

The compiler automatically substitutes the parameters you specified, in this case `int` and `20`, in place of the template formal parameters. You can create other instances of this template using other built-in types as well as user-defined types.

Function Templates

A function template defines a family of functions. To declare a function template, use the keyword `template` to define the formal parameters, which are types, then define the function in terms of those types. For example, the following is a function template for a swap function. It simply swaps the values of its two arguments:

```
template<class T>
void swap(T& val1, T& val2)
{
        T temp=val1;
        val1=val2;
        val2=temp;
}
```

The argument types to the function template `swap` are not specified. Instead, the formal parameter, `T`, is a placeholder for the types. To use the function template to create an actual function instance (a template function), you simply call the function defined by the template and provide actual parameters. A version of the function with those parameter types is automatically created.

For example, the following main program calls the function `swap` twice, passing `int` parameters in the first case and `float` parameters in the second case. The compiler uses the `swap` template to automatically create two versions, or instances, of `swap`, one that takes `int` parameters and one that takes `float` parameters.

```
void main()
{       int i=2, j=9;
        swap(i,j);

        float f=2.2, g=9.9;
        swap(f,g);
}
```

Other versions of `swap` can be created with other types that exchange the values of the given type.

Template Code is Stored in a Repository

When you declare a template, the compiler stores information about the template in a **repository**. The compiler creates a directory, `ptrepository` for "parameterized type repository," and stores information about your template there. When you use the template, the compiler automatically instantiates the template using the repository.

CC Options for Templates

You can change the default compiler behavior by using the `-pt` options. For a complete description of these options see Table 3-1 in Chapter 3 of this manual and the section "Template Instantiation User Guide" in the *C++ Language System Selected Readings*. Note that the `PTOPTS` environment variable described in this article is not supported by HP C++. Use `CXXOPTS` instead. `CXXOPTS` is described in Chapter 3.

Exception Handling

HP C++ version 3.0 added a mechanism to respond to error conditions in a controlled way. This mechanism is called exception handling. Exception conditions are error situations that occur while your program is running.

For more information about exception handling, see the *The C++ Programming Language* and the *HP C++ Release Notes*.

You Must Use the +eh Option

To use exception handling, you *must* use the `+eh` option. If your program consists of multiple source files, you must compile *all* the files in the program with the `+eh` option. If some files were compiled with `+eh` and some without, when you link with the `CC` command, `c++patch` will give an error and the files will not link.

The throw, catch, and try Statements

To signal an error condition, you "raise an exception" with the `throw` statement. To respond to the error condition, you "handle the exception" with the `catch` statement. The `throw` statement must appear either within a `try` block, which is defined by the keyword `try`, or in functions called from the `try` block. The `catch` statement must appear immediately after the `try` block.

Examples

For example, the following declares a class `Stack`, which is an integer stack of a maximum of 5 elements. The class `Stack` declares two public nested classes `Overflow` and `Underflow` which will be used for handling those error conditions. When the stack overflows or underflows, the appropriate exceptions are thrown:

```
#include <iostream.h>
const STACKMAX=5;      // Maximum size of the stack.

class Stack
{
public:

     class Overflow    // An exception class.
     { public:
             int overflowval;
             Overflow(int i) : overflowval(i) {}
     };

     class Underflow           // An exception class.
     { public:
             Underflow () {}
     }

     Stack(){top=-1;}
     void push(int item)
         {if (top<(STACKMAX-1)) thestack[++top]=item;
          else throw Overflow(item);}
     int pop()
         {if (top>-1) return thestack[top--];
          else throw Underflow();}
private:
     int thestack[STACKMAX];
     int top;
};
```

The following main program declares a stack and exception handlers for the overflow and underflow stack conditions. The program forces the stack to overflow causing the exception handler to be invoked.

```
#include <iostream.h>
void main()
{
    Stack mystack;
    int i=5, j=25;

// Here is the try block where
// exception handlers are available.

    try
    {   mystack.push(i);
        mystack.push(j);
        mystack.push(1);
        mystack.push(1234);
        mystack.push(999);

// Stack is now full.  Force an exception:

        mystack.push(50); // This will throw Stack::Overflow.
    }

// Here are the exception handlers.

    catch (Stack::Overflow& s)
    {
        cout << "Stack has overflowed trying to push: "
             << s.overflowval << endl;
    }
    catch (Stack::Underflow& s)
    {   cout << "Stack underflow has occurred." << endl; }
}
```

The above program displays the following message:

```
Stack has overflowed trying to push: 50
```

2

The HP C++ Preprocessor

Preprocessing Directives

This chapter presents information about the HP C++ preprocessor. If you are familiar with the HP C preprocessor described in the *HP C/HP-UX Reference Manual*, you may already be acquainted with some of this chapter's contents.

Overview

A **preprocessor** is a text processing program that manipulates the text within your source file. You enter **preprocessing directives** into your source file to direct the preprocessor to perform certain actions on the source file. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress the compilation of part of the file by conditionally removing sections of text. It also expands preprocessor macros and conditionally strips out comments.

The preprocessor program, `Cpp.ansi`, is invoked automatically when you compile your C++ source code. (You can use the `-Ac` option to invoke the compatibility mode preprocessor, `Cpp`.)

When the preprocessor is finished, your preprocessed C++ code is passed to the HP C++ compiler. For more information on the phases of the compiler see Chapter 3, "Compiling and Executing HP C++ Programs."

HP C++ provides two modes of preprocessor operation: *ANSI C mode* and *compatibility mode*. ANSI C mode is the default. If you want the compatibility mode preprocessor, use the `-Ac` option of the `CC` command. Refer to "Compiling HP C Programs" in the *HP C/HP-UX Reference Manual* for further information on compatibility and ANSI C modes. Refer to "Compiling and Executing HP C++ Programs" in Chapter 3 of this manual for further information on `CC` options.

Syntax

preprocessor-directive ::=
include-directive newline
macro-directive newline
conditional-directive newline
line-directive newline
pragma-directive newline
error-directive newline

Preprocessing directives control the following general functions, each of which is discussed in subsequent sections:

- source file inclusion

  You can direct HP C++ to include other source files at a given point. This is normally used to centralize declarations or to access standard system headers such as `iostream.h`.

- macro replacement

  You can direct HP C++ to replace token sequences with other token sequences. In C, this is frequently used to define names for constants rather than explicitly putting the constant value into the source file. In C++ you can also use the keyword `const` to define constants.

- conditional compilation

  You can direct HP C++ to check values and flags and to compile or skip source code based on the outcome of a comparison. This feature is useful in writing a single source that will be used for several different configurations.

- line control

  You can direct HP C++ to set the line number and file name of the next line.

- pragma directives

  You can direct HP C++ to give implementation-dependent instructions, called **pragmas**, to the compiler. Because they are system-dependent, pragmas are not portable.

- error directives

You can create diagnostic messages that will be produced by HP C++.

Using Preprocessor Directives

The following lists rules and guidelines for using preprocessor directives:

- All preprocessing directives must begin with a pound sign (**#**) as the first character on a line of your source file. (However, if you are in ANSI C mode only, white-space characters may precede the **#** character.)
- The **#** character is followed by any number of spaces and horizontal tab characters and the preprocessing directive.
- The preprocessing directive is terminated by a newline character.
- Preprocessing directives, as well as normal source lines, can be continued over several lines. End the lines that are to be continued with a backslash (\).
- Some directives can take actual arguments or values.
- Comments in the source file that are not passed through the preprocessor are replaced with a single white space character (ASCII character number decimal 32).

The following are examples of preprocessing directives:

include-directive:	`#include <iostream.h>`
macro-directive:	`#define MAC x+y`
conditional-directive:	`#ifdef MAC`
	`# define x 25`
	`# endif`
line-directive:	`#line 5 "myfile"`
pragma-directive:	`#pragma OPTIMIZE ON`
error-directive:	`#error "FLAG not defined!"`

Source File Inclusion

You can include the contents of other files within the source file prior to compilation by using the `#include` directive.

Syntax

```
include-directive ::=
     #include <filename>
     #include "filename"
     #include identifier
```

Description

The `#include` preprocessing directive causes HP C++ to read source input from the file named in the `#include` directive. Usually, include files are named:

filename.h

If the file name is enclosed in angle brackets (< >), the system directory is searched to find the named file. If the file name is enclosed in double quotation marks (" "), HP C++ searches your current directory for the specified file. Refer to "System Library and Header Files" in Chapter 3 for a detailed description of how an `#include` file is found.

Files that are included may contain `#include` directives themselves. HP C++ supports a nesting level of at least 35 `#include` files.

The arguments to the `#include` directive are subject to macro replacement before the directive processes them. Thus, if you use an `#include` directive of the form `#include` *identifier*, *identifier* must be a previously defined macro that when expanded produces one of the above defined forms of the `#include` directive. Refer to the next section, "Macro Replacement," for more information on macros.

Error messages produced by HP C++ indicate the name of the `#include` file where the error occurred, as well as the line number within the file.

Examples

```
#include <iostream.h>
#include "myheader.h"
#ifdef   MINE
#   define  filename  "file1.h"
#else
#   define  filename  "file2.h"
#endif
#include filename
```

Macro Replacement

You can define C++ macros to substitute text in your source file.

Syntax

macro-directive ::=
`#define` *identifier [replacement-list]*
`#define` *identifier([identifier-list]) [replacement-list]*
`#undef` *identifier*

replacement-list ::=
    token
    replacement-list token

Description

A `#define` preprocessing directive of the form:

`#define` *identifier [replacement-list]*

defines the *identifier* as a macro name that represents the *replacement-list*. The macro name is then replaced by the list of tokens wherever it appears in the source file (except inside of a string, character constant, or comment). A

macro definition remains in force until it is undefined through the use of the `#undef` directive or until the end of the compilation unit.

Note

The *replacement-list* must fit on one line. If the line becomes too long, it can be broken up into several lines provided that all lines but the last are terminated by a "\" character. The following is an example.

```
#define mac very very long\
replacement string
```

The "\" must be the last character on the line. You cannot add any spaces or comments after it.

Macros can be redefined without an intervening `#undef` directive. Any parameter used must agree in number and spelling with the original definition, and the replacement lists must be identical. All white space within the *replacement-list* is treated as a single blank space regardless of the number of white-space characters you use. For example, the following `#define` directives are equivalent:

```
#define foo x  +  y

#define foo x + y
```

The *replacement-list* may be empty. If the token list is not provided, the macro name is replaced with no characters.

Macros with Parameters

You can create macros that have parameters. The syntax of the `#define` directive that includes formal parameters is as follows:

`#define` *identifier([identifier-list]) [replacement-list]*

The macro name is the *identifier*. The formal parameters are provided by the *identifier-list* enclosed in parentheses. The open parenthesis must immediately follow the *identifier* with no intervening white space. If there is a space between the identifier and the parenthesis, the macro is defined as if it were the first form and the *replacement-list* begins with the "(" character.

The formal parameters to the macro are separated with commas. They may or may not appear in the *replacement-list*. When the macro is invoked, the actual arguments are placed in a parenthesized list following the macro name. Commas enclosed in additional matching pairs of parentheses do not separate arguments but are themselves components of arguments.

The actual arguments replace the formal parameters in the token string when the macro is invoked.

Specifying String Literals with the # Operator

If a formal parameter in the macro definition directive's replacement string is preceded by a **#** operator, it is replaced by the corresponding argument from the macro invocation, preceded and followed by a double-quote character (") to create a string literal. This feature, available only with the ANSI C preprocessor, may be used to turn macro arguments into strings. This feature is often used with the fact that HP C++ concatenates adjacent strings.

For example,

```
#include <iostream.h>
#define display(arg) cout << #arg << "\n"  //define the macro
main()
{
     display(any string you want to use);    //use the macro
}
```

After HP C++ expands the macro definition in the preceding program, the following code results:

```
  :
main ()
{
     cout << "any string you want to use" << "\n";
}
```

Concatenating Tokens with the ## Operator

Use the special **##** operator to form other tokens by concatenating tokens used as actual arguments. Each instance of the **##** operator is deleted and the tokens preceding and following the **##** are concatenated into a single token. If either of these names is a formal parameter of the macro, the corresponding

argument at invocation is used. This is useful in forming unique variable names within macros.

Example 1. The following illustrates the ## operator:

```
      // define the macro; the ## operator
      // concatenates arg1 with arg2
#define concat(arg1,arg2) arg1 ## arg2

main()
{
     int concat(fire,fly);
     concat(fire,fly) = 1;
     printf("%d \n",concat(fire,fly));
}
```

Preprocessing the preceding program yields the following:

```
main()
{
     int firefly ;
     firefly = 1;
     printf("%d \n",firefly );
}
```

Example 2. You can use the **#** and **##** operators together:

```
#include <iostream.h>
#define show_me(arg) int var##arg=arg;\
    cout << "var" << #arg << " is " << var##arg << "\n";
main()
{
    show_me(1);
}
```

Preprocessing this example yields the following code for the `main` procedure:

```
main()
{
    int var1=1; cout << "var" << "1" << " is " << var1 << "\n";
}
```

After compiling the code with `CC` and running the resulting executable file, you get the following results:

```
var1 is 1
```

Spaces around the `#` and `##` are optional.

Note The `#` and `##` operators are only valid when using the ANSI C mode preprocessor, which is the default preprocessor. They are not supported when using the compatibility mode preprocessor.

In both the `#` and `##` operations, the arguments are substituted as is, without any intermediate expansion. After these operations are completed, the entire replacement text is re-scanned for further macro expansions.

Using Macros to Define Constants

The most common use of the macro replacement is in defining a constant. In C++ you can also declare constants using the keyword `const`. See "Constants" in Chapter 1 for more information. Rather than explicitly putting constant values in a program, you can name the constants using macros, then use the names in place of the constants. By changing the definition of the macro, you can more easily change the program:

```
#define ARRAY_SIZE 1000
float x[ARRAY_SIZE];
```

In this example, the array `x` is dimensioned using the macro `ARRAY_SIZE` rather than the constant 1000. Note that expressions that may use the array can also use the macro instead of the actual constant:

```
for (i=0; i<<ARRAY_SIZE; ++i) f+=x[i];
```

Changing the dimension of `x` means only changing the macro for `ARRAY_SIZE`. The dimension changes and so do all of the expressions that make use of the dimension.

Other Macros

Two other macros include:

```
#define FALSE 0
#define TRUE 1
```

The following macro is more complex. It has two parameters and produces an inline expression which is equal to the maximum of its two parameters:

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

Note

Parentheses surrounding each argument and the resulting expression ensure that the precedences of the arguments and the result interact properly with any other operators that might be used with the `MAX` macro.

Because each argument to the `MAX` macro appears in the token string more than once, the actual arguments to the `MAX` macro may have undesirable side effects. The following example might not work as expected because the argument `a` is incremented two times when `a` is the maximum:

```
i = MAX(a++, b);
```

which is expanded to

```
i = ((a++) > (b) ? (a++) : (b))
```

Given the above macro definition, the statement

```
i = MAX(a, b+2);
```

is expanded to:

```
i = ((a) > (b+2) ? (a) : (b+2));
```

Examples

Following are additional macro examples.

```
// This macro tests a number and returns TRUE if
// the number is odd.  It returns FALSE otherwise.
#define isodd(n)  ( ((n % 2) == 1) ? (TRUE) : (FALSE))

// This macro skips white spaces.
#define eatspace()while((c=getc(input))==''||c=='\n'||c\
== '\t' )
```

Using Constants and Inline Functions instead of Macros

In C++ you can use named constants and inline functions to achieve results similar to using macros.

You can use `const` variables in place of macros. Refer to "Constant Data Types" in Chapter 1, "Overview of HP C++," for details.

You can also use inline functions in many C++ programs where you would have used a function-like macro in a C program. Using inline functions reduces the likelihood of unintended side effects, since they have return types and generate their own temporary variables where necessary.

Example

The following program illustrates the replacement of a macro with an inline function:

```
#include <stream.h>
#define distance1(rate,time) (rate * time)
// replaced by :
inline int distance2 ( int rate, int time )
{
     return ( rate * time );
}
int main()
{
     int i1 = 3, i2 = 3;

     printf("Distance from macro : %d\n",
             distance1(i1,i2) );
     printf("Distance from inline function : %d\n",
             distance2(i1,i2) );
}
```

Predefined Macros

In addition to `__LINE__` and `__FILE__` (refer to "Line Control" below), HP C++ provides the `__DATE__`, `__TIME__`, `__STDCPP__`, `__cplusplus` and `c_plusplus` predefined macros. Table 2-1 describes the complete set of macros that are predefined to produce special information. They may not be undefined.

Table 2-1. Predefined Macros

Macro Name	Description
`__cplusplus` `c_plusplus`	Produces the decimal constant 1, indicating that the implementation supports C++ features. You should use `__cplusplus` because `c_plusplus` will be phased out in a future release.
`__DATE__`	Produces the date of compilation in the form *Mmm dd yyyy*.
`__FILE__`	Produces the name of the file being compiled.
`__LINE__`	Produces the current source line number.
`__STDCPP__`	Produces the decimal constant 1, indicating that the preprocessor is in the ANSI C mode.
`__TIME__`	Produces the time of compilation in the form *hh:mm:ss*.

Note `__DATE__`, `__TIME__`, and `__STDCPP__` are not defined in the compatibility mode preprocessor.

Conditional Compilation

Conditional compilation directives allow you to delimit portions of code that are compiled only if a condition is true.

Syntax

conditional-directive ::=

`#if`	*constant-expression newline*
`#ifdef`	*identifier newline [group]*
`#ifndef`	*identifier newline [group]*
`#else`	*newline [group]*
`#elif`	*constant-expression newline [group]*
`#endif`	

Note `#elif` is available only with the ANSI C preprocessor.

Here, *constant-expression* may also contain the `defined` operator:

`defined` *identifier*
`defined (`*identifier*`)`

Description

You can use `#if`, `#ifdef`, or `#ifndef` to mark the beginning of the block of code that will only be compiled conditionally. An `#else` directive optionally sets aside an alternative group of statements. You mark the end of the block using an `#endif` directive.

The following `#if` directive illustrates the structure of conditional compilation:

```
#if constant-expression
      ⋮
(Code that compiles if the expression evaluates to a nonzero value.)
      ⋮
#else
      ⋮
(Code that compiles if the expression evaluates to zero.)
      ⋮
#endif
```

The *constant-expression* is like other C++ integral constant expressions except that all arithmetic is carried out in `long int` precision. Also, the expressions cannot use the `sizeof` operator, a cast, an enumeration constant, or a `const` object.

Using the defined Operator

You can use the `defined` operator in the `#if` directive to use expressions that evaluate to 0 or 1 within a preprocessor line. This saves you from using nested preprocessing directives.

The parentheses around the identifier are optional. Below is an example:

```
#if defined (MAX) & ! defined (MIN)
      ⋮
```

Without using the `defined` operator, you would have to include the following two directives to perform the above example:

```
#ifdef max
#ifndef min
```

Using the #if Directive

The `#if` preprocessing directive has the form:

`#if` *constant-expression*

Use `#if` to test an expression. HP C++ evaluates the expression in the directive. If the expression evaluates to a nonzero value (TRUE), the code following the directive is included. Otherwise, the expression evaluates to FALSE and HP C++ ignores the code up to the next `#else`, `#endif`, or `#elif` directive.

All macro identifiers that appear in the *constant-expression* are replaced by their current replacement lists before the expression is evaluated. All `defined` expressions are replaced with either 1 or 0 depending on their operands.

The #endif Directive

Whichever directive you use to begin the condition (`#if`, `#ifdef`, or `#ifndef`), you must use `#endif` to end the *if* section.

Using the #ifdef and #ifndef Directives

The following preprocessing directives test for a definition:

`#ifdef` *identifier*
`#ifndef` *identifier*

They behave like the `#if` directive, but `#ifdef` is considered true if the *identifier* was previously defined using a `#define` directive or the `-D` option. `#ifndef` is considered true if the *identifier* is not yet defined.

Nesting Conditional Compilation Directives

You can nest conditional compilation constructs. Delimit portions of the source program using conditional directives at the same level of nesting, or with a `-D` option on the command line.

Using the #else Directive

Use the `#else` directive to specify an alternative section of code to be compiled if the `#if`, `#ifdef`, or `#ifndef` conditions fail. The code after the `#else` directive is included if the code following any of the `#if` directives is not included.

Using the #elif Directive

The `#elif` *constant-expression* directive, available only with the ANSI C preprocessor, tests whether a condition of the previous `#if`, `#ifdef`, or `#ifndef` was false. `#elif` has the same syntax as the `#if` directive and can be used in place of an `#else` directive to specify an alternative set of conditions.

Examples

The following examples show valid combinations of these conditional compilation directives:

```
#ifdef SWITCH        // compiled if SWITCH is defined
#else                // compiled if SWITCH is undefined
#endif               // end of if

#if defined(THING)   // compiled if THING is defined
#endif               // end of if

#if A>47             // compiled if A is greater than 47
#else
#if A < 20           // compiled if A is less than 20
#else                // compiled if A is greater than or equal
                     // to 20 and less than or equal to 47
#endif               // end of if, A is less than 20
#endif               // end of if, A is greater than 47
```

The following are more examples showing conditional compilation directives:

```
#if (LARGE_MODEL)
#define INT_SIZE 32     // Defined to be 32 bits.
#elif defined (PC) & defined (SMALL_MODEL)
#define INT_SIZE 16     // Otherwise, if PC and SMALL_MODEL
                        // are defined, INT_SIZE is defined
                        // to be 16 bits.
#endif

#ifdef DEBUG            // If DEBUG is defined, display
cout << "table element : \n";  // the table elements.
for (i=0; i << MAX_TABLE_SIZE; ++i)
     cout << i << " " << table[i] << '\n';
#endif
```

Line Control

You can cause HP C++ to set line numbers during compilation from a number specified in a line control directive. (The resulting line numbers appear in error message references, but do not alter the line numbers of the actual source code.)

Syntax

line-directive `::=`
`#line` *digit-sequence [filename]*

Description

The `#line` preprocessing directive causes HP C++ to treat lines following it in the program as if the name of the source file were *filename* and the current line number were *digit-sequence*. This serves to control the file name and line number that are given in diagnostic messages. This feature is used primarily by preprocessor programs that generate C++ code. It enables them to force HP C++ to produce diagnostic messages with respect to the source code that is input to the preprocessor rather than the C++ source code that is output.

HP C++ defines two macros that you can use for error diagnostics. The first is `__LINE__`, an integer constant equal to the value of the current line number. The second is `__FILE__`, a quoted string literal equal to the name of the input source file. You can change `__FILE__` and `__LINE__` using `#include` or `#line` directives.

Example

```
#line 5 "myfile"
```

Pragma Directive

A `#pragma` directive is an instruction to the compiler. You typically use a `#pragma` directive to control the actions of the compiler in a particular portion of a program without affecting the program as a whole.

Syntax

pragma-directive ::=
    `#pragma` *[token-list]*

Description

The `#pragma` directive is ignored by the preprocessor, and instead is passed on to the C++ compiler. It provides implementation-dependent information to HP C++. Refer to Chapter 3, "Compiling and Executing HP C++ Programs," for descriptions of pragmas recognized by HP C++. Any pragma that is not recognized by HP C++ will generate a warning from the compiler. The following is an example of a `#pragma` directive.

Example

```
#pragma OPTIMIZE ON
```

Error Directive

Syntax

error-directive ::=
    `#error` *[preprocessor tokens]*

Description

The `#error` directive causes a diagnostic message, along with any included token arguments, to be produced by HP C++.

Examples

```
     // This directive will produce the diagnostic
     // message "FLAG not defined!".
#ifndef FLAG
#error "FLAG not defined!"
#endif

     // This directive will produce the diagnostic
     // message "TABLE_SIZE must be a multiple of 256!".
#if TABLE_SIZE % 256 != 0
#error "TABLE_SIZE must be a multiple of 256!"
#endif
```

Note The `#error` directive is not supported when using the compatibility mode preprocessor.

Trigraph Sequences

Description

The C++ source code character set is a *superset* of the ISO 646-1983 Invariant Code Set. To enable you to use only the reduced set, you can use **trigraph sequences** to represent those characters *not* in the reduced set. A trigraph sequence is a set of three characters that is replaced by a corresponding single character. The preprocessor replaces all trigraph sequences with the corresponding character. Table 2-2 gives the complete list of trigraph sequences and their replacement characters.

Example

The line below contains the trigraph sequence `??=`:

```
??=line 5 "myfile"
```

When this line is compiled it becomes:

```
#line 5 "myfile"
```

Table 2-2. Trigraph Sequences and Replacement Characters

Trigraph Sequence	Replacement
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

3

Compiling and Executing HP C++ Programs

This chapter describes how to compile and execute HP C++ programs on the HP-UX operating system. It presents the `CC` command and its options, which allow you to access the compiling system. You can compile HP C++ programs into C, assembly, object, or executable files. Optionally, you can optimize the code.

The chapter is organized into the following topics:

- phases of the compiling system
- compiling with `CC`
- system library and header files
- creating and using shared libraries
- a complete example C++ program
- executing HP C++ programs

The chapter concludes with a programming example illustrating the concept of object-oriented program development.

Phases of the Compiling System

When you compile an HP C++ program it passes through one or more phases or subprocesses controlled by a component of the compiling system. The `CC` command invokes the components of the HP C++ compiling system automatically when you use the `CC` command. You do not have to invoke each component yourself.

Use the `-v` and `-ptv` options to see detailed information about each component as it executes. The following sections describe these phases and components.

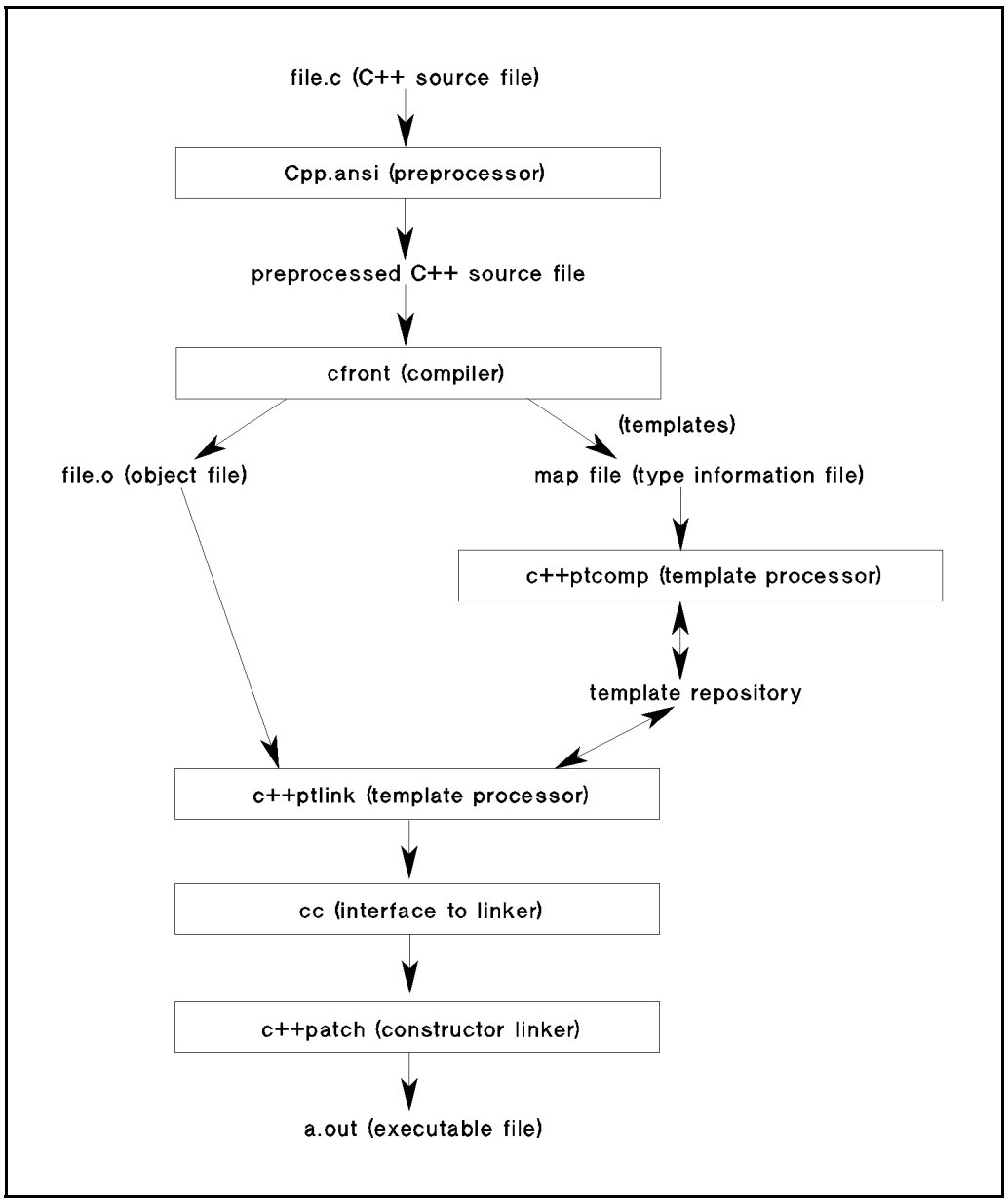


Figure 3-1. Phases of the HP C++ Compiling System in Compiler Mode

What Happens in Compiler Mode

This section describes the compiler phases or subprocesses that execute when you compile a C++ program in **compiler mode**, the default mode. In compiler mode your C++ source code is compiled directly to object code. Refer to Figure 3-1.

In **translator mode**, your C++ code is translated to C code, then compiled by the C compiler. See Figure 3-2 for more information on translator mode.

Preprocessing

When you compile a C++ source program using either compiler mode or translator mode, HP C++ invokes the preprocessor `Cpp.ansi` on your programs that have the file name suffix `.c` or `.C`. The preprocessor examines all lines beginning with a `#`, 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, compiling with `cfront`, is successful, the `.i` file in the temporary directory is deleted by default. Use the `-P` option to save the `.i` files.

For more information on the preprocessor, see Chapter 2, "The HP C++ Preprocessor."

Compiling C++ Source Code

When you use the default compiler mode, the compilation phase runs `cfront` in compiler mode. `cfront` compiles the preprocessed C++ code and generates object code in the `.o` file.

`cfront` also creates a map file, a temporary file containing information about the data types in your code.

In compiler mode the C++ code is *not* translated to C code.

Compile-Time Template Processing

The compile-time template processing phase runs `c++ptcomp` which merges the map file into the repository.

Link-Time Template Processing

The link-time template processing phase runs `c++ptlink` and retrieves information about templates from the repository to automatically instantiate templates. `c++ptlink` may create additional object files in the repository. This phase is entered only if templates need to be instantiated.

Linking

In the link phase, the `CC` command invokes the linker, `/bin/ld`, using the `cc` interface. The linker produces an executable program that includes the start-up routines from `/lib/crt0.o` (`/opt/langtools/lib/*crt0.o` for Versions A.10.01 and later) and any needed library routines from the archive libraries `/lib/libc.a`, `/usr/lib/libC.a`, and `/usr/lib/libC.ansi.a`, or references to library routines from the shared libraries `/lib/libc.sl`, `/usr/lib/libC.sl`, and `/usr/lib/libC.ansi.sl`. If you are using exception handling, the libraries in `/usr/lib/CC/eh` are used.

External references are resolved, libraries are searched to resolve references to library routines, and the object files are combined into an executable program file, `a.out` by default.

Linking Constructors and Destructors

The patch phase runs `c++patch`. `c++patch` links or chains constructors and destructors of nonlocal static objects in the executable file or shared library.

By default, the name of the executable file is `a.out`.

What Happens in Translator Mode

This section describes the compiler phases or subprocesses that execute when you compile a C++ program in **translator mode**. In translator mode, your C++ code is translated to C code, then compiled by the C compiler. Refer to Figure 3-2.

Translator mode is used only when you use the `+T` option to `CC`. By default, C++ uses **compiler mode**. See Figure 3-1 for more information on compiler mode.

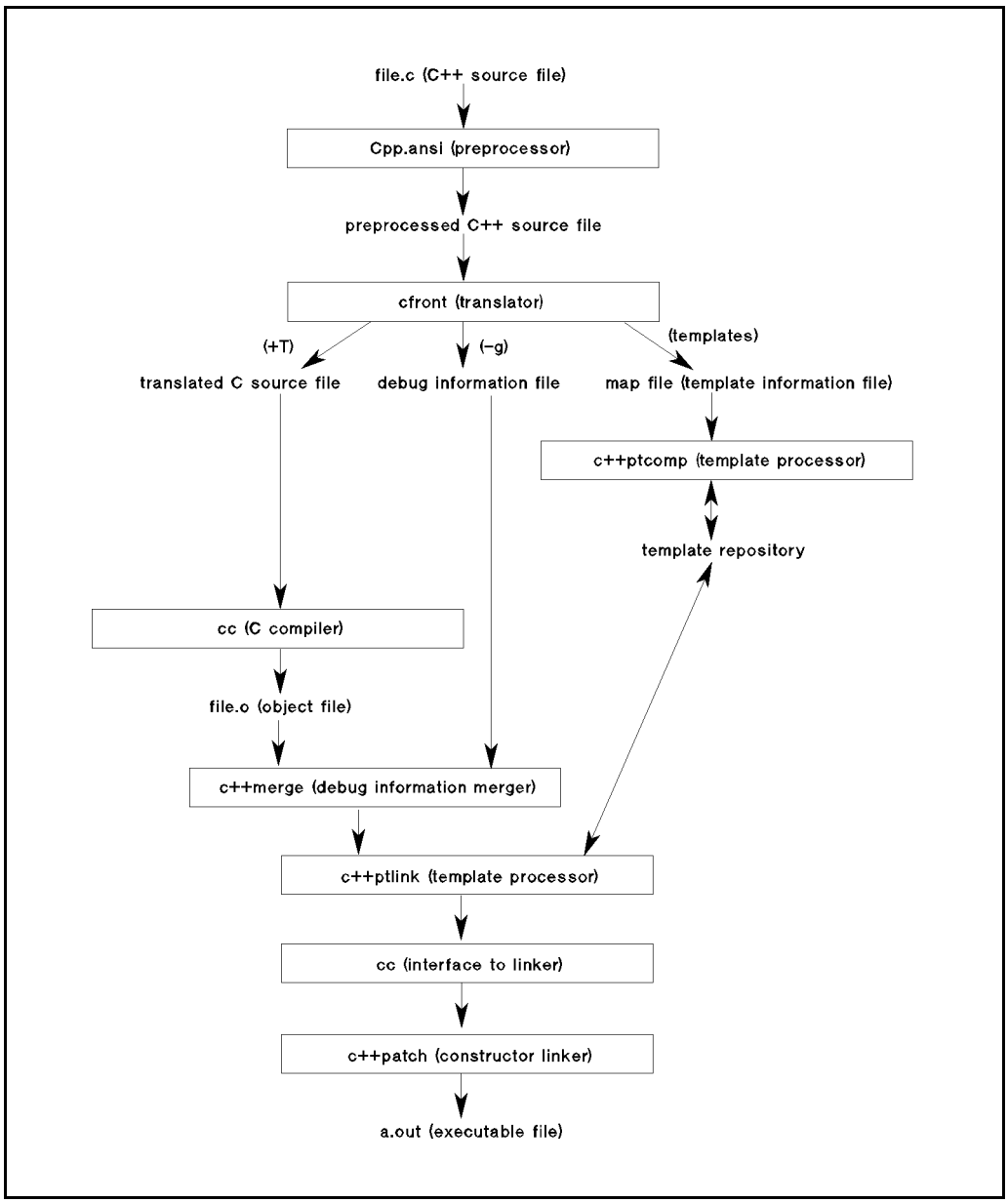


Figure 3-2. Phases of the HP C++ Compiling System in Translator Mode

3-6 Compiling and Executing HP C++ Programs

Preprocessing

When you compile a C++ source program using translator mode, HP C++ invokes the preprocessor `Cpp.ansi` on your programs the same as it does in compiler mode and produces a preprocessed version of your program with the file name suffix `.i`.

For more information on the preprocessor, see Chapter 2, "The HP C++ Preprocessor."

Translating C++ Source Code to C

The translation phase runs `cfront` in **translator mode**. `cfront` takes the output of the preprocessor (the `.i` files containing C++ source code), performs syntax and type checking, and translates HP C++ source programs to C source programs. The temporary C files created by `cfront` are exact translations of the C++ code ready for the HP C compiler to compile.

In addition, `cfront` creates a map file, a temporary file containing information about the data types in your code.

If you specified the `-g` or `-g1` option, `cfront` also creates a temporary file containing information for the symbolic debugger.

Compile-Time Template Processing

The compile-time template processing phase runs `c++ptcomp` which merges the map file into the repository.

Compiling the Translated C Source Code

In **translator mode** the compilation phase runs the C compiler, `cc`, which compiles the translated C source code and generates object code in the `.o` file.

Adding Debug Information

When you use **translator mode** and you compile with either the `-g` or `-g1` option, your files go through the merge phase. This phase runs `c++merge` and merges the debug information from the temporary file into the object file so you can use the symbolic debugger.

When you use compiler mode and specify either `-g` or `-g1` the HP C++ compiler adds the debug information directly to the object file.

Link-Time Template Processing

The link-time template processing phase runs `c++ptlink` and retrieves information about templates from the repository to automatically instantiate templates. `c++ptlink` may create additional object files in the repository. This phase is entered only if templates need to be instantiated.

Linking

In the link phase, the `CC` command invokes the linker, `/bin/ld`, using the `cc` interface. The linker produces an executable program that includes the start-up routines from `/lib/crt0.o` (`/opt/langtools/lib/*crt0.o` for Versions A.10.01 and later) and any needed library routines from the archive libraries `/lib/libc.a`, `/usr/lib/libC.a`, and `/usr/lib/libC.ansi.a`, or references to library routines from the shared libraries `/lib/libc.sl`, `/usr/lib/libC.sl`, and `/usr/lib/libC.ansi.sl`. If you are using exception handling, the libraries in `/usr/lib/CC/eh` are used.

External references are resolved, libraries are searched to resolve references to library routines, and the object files are combined into an executable program file, `a.out` by default.

Linking Constructors and Destructors

The patch phase runs `c++patch`. `c++patch` links or chains constructors and destructors of nonlocal static objects in the executable file or shared library.

By default, the name of the executable file is `a.out`.

Compiling with the CC Command

Use the `CC` command to invoke the HP C++ compiling system. The `CC` command invokes a driver program that runs the phases of the compiling system according to the file names and command line options that you specify.

Setting Your Path to the CC Command

The `CC` command is normally installed in the directory `/usr/bin` (`/opt/CC/bin` for HP-UX 10.*x* C++ versions). So that you can use the `CC` command, you should ensure that your `PATH` environment variable includes this directory. You can do this with the following Bourne or Korn shell commands:

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

You should modify the command that sets the `PATH` variable in the appropriate shell script file, either `.profile` or `.login`, in your home directory.

Syntax

The `CC` command has the following format:

`CC [` *options* `|` *files* `]`

where:

options — is zero or more compiler options and their arguments, if any. Single-character options that do not accept additional arguments can be grouped under either a single minus or plus sign.

files — is one or more path names, separated by blanks. Each file is either a source file, a preprocessed source file, an assembly language source file, an object file, or a library file.

Specifying Files to the CC Command

HP C++ source files must be named with extensions beginning with either `.c` or `.C`, possibly followed by additional characters. If you compile only, each HP C++ source file produces an object file with the same name as the source file, except that the extension beginning with `.c` or `.C` is changed to a `.o` extension. However, if you compile and link a single source file into an HP C++ program in one step, the `.o` file is automatically deleted.

Caution While file extensions other than `.c` or `.C` are permitted for portability from other systems, it is recommended that your source files have extensions of `.c` and `.C` only, without additional characters. Other endings may not be supported by HP tools and environments.

Files with names ending in `.i` are assumed to be preprocessor output files (refer to the `-P` compiler option). Files ending in `.i` are processed the same as `.c` or `.C` files, except that the preprocessor is not run on the `.i` file before the file is compiled.

Files with names ending in `.s` are assumed to be assembly source files. The compiler invokes the assembler to produce `.o` files from these.

Files with `.o` extensions are assumed to be relocatable object files that are to be included in the linking. All other files are passed directly to the linker by the compiler.

Unless you use the `-o` option to specify otherwise, all files that the `CC` compiling system generates are put in the working directory, even if the source files came from other directories.

Specifying Options to the CC Command

The `CC` interface supports several options that you can use to control the operation of the compiling system. You can specify these options on the `CC` command line before, after, or interspersed with file arguments.

The `CC` options have one of the two prefixes, `-` or `+`.

Each compiler option has the following format:

-optionname [optionarg]

or

+optionname [optionarg]

where:

optionname is the name of a compiler option

and

optionarg is the argument to *optionname*.

See also "The CXXOPTS Environment Variable" in this chapter for another way of specifying options to the `CC` command.

An Example of Using a Compiler Option

By default, the `CC` command names the executable file `a.out`. For example, given the following command line,

```
CC demo.C
```

the executable file is named `a.out`, just as is the case in compiling a C program with the `cc` command.

You can use the `-o` option to override the default name of the executable file produced by `CC`. For example, suppose `my_source.C` contains C++ source code and you want to create an executable file named `my_executable`. Then you would use the following command line:

```
CC -o my_executable my_source.C
```

Concatenating Options

You can concatenate some options to the `CC` command under a single prefix. The longest substring that matches an option is used. Only the last option can take an argument. You can concatenate option arguments with their options if the resulting string does not match a longer option.

For example, suppose you want to compile `my_file.C` using the options `-v`, `-g1`, and `-DPROG=sub`. Following are a few examples of command lines you could use:

```
CC my_file.C -v -g1 -DPROG=sub
CC my_file.C -vg1 -D PROG=sub
CC my_file.C -vg1DPROG=sub
CC -vg1DPROG=sub my_file.C
```

HP C++ Compiler Options

Table 3-1 lists the options HP C++ supports. Also refer to the *HP C++ Online Programmer's Guide* for the most detailed and current information.

Table 3-1. The CC Command Options

Option	Effect of Specifying the Option
`-A`*level*	Allows you to select the mode of preprocessor operation. *Level* can be either `a` or `c`: `a` requests the ANSI mode HP C++ preprocessor, `Cpp.ansi`. This is the default. `c` requests the compatibility mode HP C++ preprocessor, `Cpp`.
`-b`	Creates a shared library rather than an executable file. The object files must have been created with the `+z` or `+Z` option to generate position-independent code (PIC). For more information on shared libraries, see "Creating and Using Shared Libraries" in this chapter, and the manual *HP-UX Linker and Libraries Online User Guide*.
`-c`	Compiles one or more source files but does not enter the linking phase. The compiler produces an object file (a file ending with `.o`) for each source file (a file ending with `.c`, `.C`, `.s`, or `.i`). Note that you must eventually link object files before they can be executed.
`-C`	Prevents the preprocessor from stripping comments from your source file; comments are retained. Refer to the description of `Cpp` in the *HP-UX Reference Manual* for details.
`-D`*name=def* `-D`*name*	Defines *name* to the preprocessor `Cpp`, as if defined by the preprocessing directive `#define`. If no *=def* is given, the name is defined as "1". Refer to the `Cpp` description in the *HP-UX Reference Manual* for details.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`-depth`	Instructs the runtime system to traverse the shared library list in a depth-first manner when calling static constructors. The default is to traverse the shared libraries in a left-to-right order when calling static constructors. Use `-depth` when linking your program with the `CC` command. The order of execution of static constructors within each shared library is not affected by this option. The routine to load a shared library, `cxxsh_load()`, also traverses dependent libraries in depth-first order when the program is linked with `-depth`. For example: `CC -depth prog.C lib1.sl lib2.sl lib3.sl` compiles `prog.C`, links to the shared libraries `lib1.sl`, `lib2.sl`, and `lib3.sl`, and instructs the runtime startup code to execute the static constructors in `lib3.sl` first, `lib2.sl` next, and `lib1.sl` last. (The default order would be `lib1.sl`, then `lib2.sl`, then `lib3.sl`.)
`-E`	Runs preprocessor only on the named HP C++ or assembly programs and sends the result to standard output (`stdout`). See also the *-.suffix* option.
`-F`	Runs only `Cpp` and the HP C++ translator (see the `+T` option) on the C++ source files and sends the resulting C source code to standard output (`stdout`). See also the *-.suffix* option.
`-Fc`	Same as the `-F` option, but the output is C source code suitable to be redirected to a `.c` file that can later be compiled using `cc`. This option is equivalent to using the `-F` and the `+L` options together. See also the *-.suffix* option.
`-.`*suffix*	Causes the HP C++ translator to direct output from either the `-E`, `-F`, or `-Fc` option into a file with the corresponding *.suffix* instead of into a corresponding `.c` file. Note that *.suffix* may not be the same as the original source file *.suffix*.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`-g`	Causes the compiler to generate additional information needed by the symbolic debugger. Note, for 10.*x* and later releases you can use this option for limited debugging of optimized code with the HP/DDE debugger. To suppress expansion of inline functions use the `+d` option. See also the `-g1` option. For more information about HP Symbolic Debugger, see the *HP-UX Symbolic Debugger User's Guide*.
`-g1`	This option is the same as the `-g` option, except the compiler generates less information about your program for the symbolic debugger, thereby decreasing the size of your object file. Specifically, the `-g` option emits full debug information about every class referenced in a file, which can result in much redundant information. The `-g1` option, on the other hand, emits only a subset of this debug information. If you compile your entire application with `-g1` no debugger functionality is lost. Use `-g1` when ■ You are compiling your *entire* application with debug on and your application is large, for example, greater than 1 megabyte. Use `-g` when *either* of the following is true: ■ You are compiling only a portion of your application with debug on. ■ You are compiling you entire application with debug on and your application is not very large, for example, less than 1 megabyte. If you compile part of an application with `-g1` and part with debug off, the resulting executable may not contain complete debug information. You will still be able to run the executable, but in the debugger some classes may appear to have no members. For more information about HP Symbolic Debugger, see the *HP-UX Symbolic Debugger User's Guide*.
`-G`	Prepares the object file for profiling with `gprof++`. Refer to the online man page of `gprof++` and to the `gprof` description in the *HP-UX Reference Manual* for details.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`-I`*dir*	Adds *dir* to the directories to be searched for `#include` files by the preprocessor. For `#include` files that are enclosed in double quotes (`" "`) and do not begin with a /, the preprocessor first searches the directory of the file containing the `#include`, then the directory named in the `-I` option, and finally the standard include directories `/usr/include/CC` and `/usr/include`. For `#include` files that are enclosed in angle brackets (`< >`), the search path begins with the directory named in the `-I` option and is completed in the standard include directories `/usr/include/CC` and `/usr/include`. The current directory is not searched.
`-l`*x*	Causes the linker to search the libraries `/lib/lib`*x*`.sl` or `/lib/lib`*x*`.a`, then `/usr/lib/lib`*x*`.sl` or `/usr/lib/lib`*x*`.a` (just `/usr/lib/lib`*x*`.sl` or `/usr/lib/lib`*x*`.a` for Versions 10.*x* and later) in an attempt to resolve unresolved external references. The `-a` linker option determines whether the archive (`.a`) or shared (`.sl`) version of a library is searched. The linker searches the shared version of a library by default. Because a library is searched when its name is encountered, placement of a `-l` 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. Refer to the description of `ld` in the *HP-UX Reference Manual* for details.
`-L`*dir*	Causes the linker to search for libraries in the directory *dir* before using the default search path. This option is passed directly to the linker. The `-L` option must precede any `-l`*x* option entry on the command line; otherwise `-L` is ignored.
`-n`	Causes the program file produced by the linker to be marked as sharable. For details and system defaults, refer to the description of `ld` in the *HP-UX Reference Manual*.
`-N`	Causes the program file produced by the linker to be marked as unsharable. For details and system defaults, refer to the `ld` description in the *HP-UX Reference Manual*.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`-o` *outfile*	Causes the output of the compilation sequence to be placed in *outfile*. Without this option the default name is `a.out`. When compiling a single source file with the `-c` option, you may use the `-o` option to specify the name and location of the object file.
`-O`	Invokes the optimizer to perform level 2 optimizations. You can set other optimization levels by using the `+O` option. Refer to Chapter 4, "Optimizing HP C++ Programs", for more information about optimization.
`-P`	Preprocess only on files named on the command line without invoking further phases, leaving the result in the corresponding files with the suffix `.i`.
`-pta`	Instantiates an entire template, rather than only those members that are needed. For more information, see the "Template Instantiation User Guide" in the *C++ Language System Selected Readings*.
`-ptb`	Direct the template instantiation system to invoke `ld` instead of `nm` to do simulated linking. Using this option may slow the instantiation process considerably. You must use this option when building shared libraries that depend on other shared libraries that contain templates. It can be used for any case.
`-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`.) Use this option if your version of HP-UX has not been upgraded to support long file names. HP C++ creates template instantiation files using long file names by default. See *convertfs(1M)* 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, HP C++ uses the following list of extensions: `".h .H .hxx .HXX .hh .HH .hpp"`.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`-ptn`	Performs template instantiation at link time rather than at compile time. This option only affects programs consisting of one file, which have instantiation performed at compile time by default. Instantiation is done at link time for programs consisting of multiple files. For more information, see the "Template Instantiation User Guide" in the *C++ Language System Selected Readings*.
`-ptr`*pathname*	Specifies a repository to hold information about your templates. The information in the repository is used whenever a template is instantiated. The default repository is `./ptrepository`. If several repositories are given, only the first is writable. For more information, see the "Template Instantiation User Guide" in the *C++ Language System Selected Readings*.
`-pts`	Causes instantiations to be split into separate object files, with one function per object file. Also causes all class static data and virtual functions to be grouped into a single object file. For more information, see the "Template Instantiation User Guide" in the *C++ Language System Selected Readings*.
`-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 for source files with these extensions in the order the extensions are listed. For example, `-ptS".c .C"` specifies that template definition files can have extensions of `.c` or `.C`. By default, HP C++ uses the following list of extensions: `".c .C .cxx .CXX .cc .CC .cpp"`.
`-ptv`	Gives verbose progress reports on the instantiation process. This option is useful for understanding how templates are instantiated. For more information, see the "Template Instantiation User Guide" in the *C++ Language System Selected Readings*.
`-q`	Causes the output file from the linker to be marked as demand-loadable. For details and system defaults, see the description of `ld` in the *HP-UX Reference Manual*.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`-Q`	Causes the program file from the linker to be marked as demand-loadable. For details and system defaults, see the description of `ld` in the *HP-UX Reference Manual*.
`-s`	Causes the executable program file created by the linker to be stripped of symbol table information. Specifying this option prevents using a symbolic debugger on the resulting program. Refer to the description of `ld` in the *HP-UX Reference Manual* for more details.
`-S`	Compiles the named HP C++ program and leaves the assembly language output in a corresponding file with an `.s` suffix.
`-t`*x, name*	Substitutes or inserts subprocess *x* using *name*, where *x* is one or more identifiers indicating the subprocess or subprocesses. This option works in two modes: 1) if *x* is a single identifier, *name* represents the full path name of the new subprocess; 2) 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. The value of *x* can be one or more of the following: **Value** — **Description** `a` — Assembler (standard suffix is `as`). `b` — The C compiler driver (`cc`) used to compile the translated C++ code and invoke the assembler and the linker. `c` — The C compiler (translator mode only; standard suffix is `ccom`.) `C` — C++ compiler (standard suffix is `cfront`). `f` — Filter tool (`c++filt`). `l` — Linker (standard suffix is `ld`). `m` — Merge tool (`c++merge`; translator mode only). `0` (zero) — Same as `c`. See also Table 3-2. `p` — Preprocessor (standard suffix is `Cpp`). `P` — Patch tool (`c++patch`). `x` — All subprocesses.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`-U` *name*	Removes (undefines) any initial definition of *name* in the preprocessor. Refer to the `Cpp` description in the *HP-UX Reference Manual* for details.
`-v`	Enables the verbose mode, sending a step-by-step description of the compilation process to `stderr`. This is especially useful for debugging or for learning the appropriate commands for processing a C++ file.
`-w`	Suppresses warning messages.
`-W`*x,arg1 [,arg2,..,argn]*	Passes the arguments *arg1* through *argn* to the subprocess *x* of the compilation; *x* can be one of the values described under the `-t`*x, name* option with the addition of `d`, to pass an option to the `CC` command.
`-Y`	Enables Native Language Support (NLS) of 8-bit and 16-bit, (also 4-byte EUC for HP-UX 10.*x* C++ versions) characters in comments, string literals, and character constants. Refer to `hpnls`, `lang`, and `environ` in the *HP-UX Reference Manual* for a description of the NLS model. The language value (refer to `environ` for the `LANG` environment variable) is used to initialize the correct tables for interpreting comments, string literals, and character constants. The language value is also used to build the path name to the proper message catalog.
`-y`	Enable the storage of static analysis information in the generated object files. This information can be used by the static analysis tool which is part of the HP SoftBench software development environment.
`-Z`	Allows dereferencing of null pointers at run time. The value returned from a dereferenced null pointer is zero.
`-z`	Disallows dereferencing of null pointers at run time. Fatal errors result if null pointers are dereferenced.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`+a{0|1}`	Specifies which style of declarations to produce. In translator mode, the compiler can generate either ANSI C or "Classic C" (also known as K&R C, for Kernighan and Ritchie, authors of a book on the C language) style declarations. The `+a0` option, the default, causes the translator to produce "Classic C" style declarations. The `+a1` option causes the translator to produce ANSI C style declarations. When you use the `+a0` option in compiler mode, value parameters of type `float` are promoted to type `double`. When you use `+a1`, `float` parameters are not promoted, but are passed as type `float`. This maintains compatibility with translator mode.
`+d`	Prevents the expansion of inline functions. This option is useful when you are debugging your code because you cannot set breakpoints at inline functions. This option defeats inlining thereby allowing you to set breakpoints at functions specified as inline.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`+DA`*architecture*	Generates object code for a particular version of the PA-RISC architecture. Also specifies which version of the HP-UX math library to link in when you have specified `-lm` (`-lm` or `-lM` for HP-UX 9.x only). See the *HP-UX Floating-Point Guide* for more information about using math libraries. *architecture* can be either a model number of an HP 9000 system (such as 730 or 877), or one of the PA-RISC architecture designations `1.0`, `1.1`, or `2.0` (`2.0` is available for versions 10.22 and later). For example, specifying `+DA1.1` or `+DA867` generates code for the PA-RISC 1.1 architecture. See the file `/usr/lib/sched.models` for model numbers and their architectures. (Refer to the file `/opt/langtools/lib/sched.models` for Versions A.10.01 and later.) Use the command `uname -m` to determine the model number of your system. Object code generated for PA-RISC 1.1 will *not* execute on PA-RISC 1.0 systems. For versions 10.22 and later note that object code generated for PA-RISC 2.0 will *not* execute on PA-RISC 1.1 systems: To generate code compatible across PA-RISC 1.1 and 2.0 workstations and servers, use the `+DAportable` option. If you are using version A.10.22 or later, do not use the `+DA1.0` option. For best performance use `+DA` with the model number or architecture where you plan to execute the program. Beginning with the HP-UX 10.20 release, the default object code generated by HP compilers is determined automatically as that of the machine on which you compile. (Previously, the default code generation was PA-RISC 1.0 on all Series 800 servers and PA-RISC 1.1 on Series 700 workstations.)

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`+DS`*model*	Performs instruction scheduling tuned for a particular implementation of the PA-RISC architecture. *model* can be either a model number of an HP 9000 system (such as 730 or 877), or one of the PA-RISC implementation designations `1.0`, `1.1`. or `2.0` (`2.0` is available for versions 10.22 and later) For example, specifying `+DS720` performs instruction scheduling tuned for one implementation of PA-RISC 1.1. Specifying `+DS745` performs instruction scheduling for another implementation of PA-RISC 1.1. Specifying `+DS1.0`, `+DS1.1`, or `+DS2.0` performs scheduling for a representative PA-RISC 1.0, 1.1, or 2.0 system, respectively. To improve performance on a particular model of the HP 9000, use `+DS` with that model number. See the file `/usr/lib/sched.models` for model numbers and their architectures. (Refer to the file `/opt/langtools/lib/sched.models` for Versions A.10.01 and later.) Use the command `uname -m` to determine the model number of your system. Object code with scheduling tuned for a particular model *will* execute on other HP 9000 systems, although possibly less efficiently. If you do not specify this option, the default instruction scheduling is for the system you are compiling on.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`+[no]dup_ static_ removal`	(Versions 10.22 and later.) Removes duplicate inline member functions that were dropped out of line because they were too complex or too large. The removal will reduce the size of executables and enhance performance. The default, `+nodup_static_removal`, does not remove duplicate static member functions. Note: Use of the option `+dup_static_removal` may give you the linker error: `Common block requests for` *functionname* `have different lengths`. You will get this error in one of two cases. One, your code violates the C++ requirement that "all inline member functions with the same name must also have the same body." Two, you use different compiler options to compile the duplicate inline member functions of different compilation units. Library providers who ship header files may not want to use `+dup_static_removal` because they may not know if their users compile with the same options as they do. For example: `CC +dup_static_removal prog.C` removes duplicate static member functions.
`+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 this module, that is initialized. When neither option is used, virtual tables are static, that is, 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 these class definitions. Refer to the note on the next page for more information.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`+eh`	Enables exception handling. To use exception handling, you *must* use this option on *all* of the files in your program. If some files have been compiled with this option and some have not, when you link with the `CC` command, `c++patch` will give an error and the files will not link.
`+ESfic`	(Versions 10.22 and later) Replaces millicode calls with inline fast indirect calls. The `+ESfic` compiler option affects how function pointers are dereferenced in generated code. The default is to generate low-level millicode calls for function pointer calls. The `+ESfic` option generates code that calls function pointers directly, by branching through them. Note: The `+ESfic` option should only be used in an environment where there are no dependencies on shared libraries. The application must be linked with archive libraries only. Using this option can improve run-time performance.
`+help`	(Versions 10.22 and later) Invokes the initial menu window of the *HP C++ Online Programmer's Guide*. If `+help` is used on any command line, the compiler invokes the online reference and then processes any other arguments. If `$DISPLAY` is set, `+help` invokes the `helpview` command. If the display variable is not set, a message so indicates. For example: `CC +help` invokes the online programmer's guide.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option

`+I`	Instructs the compiler to instrument the object code for collecting run-time profile data. The profiling information can then be used by the linker to perform profile-based optimization. Code generation and optimization phases are delayed until link time by this option.
	
	After compiling and linking with `+I`, run the resultant program using representative input data to collect execution profile data. Finally, relink with the `+P` option to perform profile-based optimization.
	
	Profile data is stored in `flow.data` by default. See the `+df`*name* option for information on controlling the name and location of this data file.
	
	This option is incompatible with `+eh`, `-g`, `-g1`, `-G`, `+P`, `-s`, `-S`, and `-y`.
	
	For example:
	
	`CC +I -O -c prog.C`
	`CC +I -O -o prog.pbo prog.o`
	
	compiles `prog.C` with optimization, prepares the object code for data collection, and creates the executable file `prog.pbo`. Running `prog.pbo` collects run-time information in the file `flow.data` in preparation for optimization with `+P`.

`+i`	Causes an intermediate C language source file with the suffix `..c` to be produced in the current directory. This option is only valid with the `+T` option.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`+k`	(For HP-UX 10.*x* C++ versions.) By default, the HP C++ compiler generates short-displacement code sequences for programs that reference global data in shared libraries. For nearly all programs this is sufficient. If your program references a large amount of global data in shared libraries, the default code generation for referencing that global data may not be sufficient. If this is the case, when you link your program the linker gives an error message indicating that you need to recompile with the `+k` option. The `+k` option generates long-displacement code sequences so a program can reference large amounts of global data in shared libraries. Use `+k` only when the linker generates a message indicating you need to do so. For example: `CC +k prog.C mylib.sl` Compiles `prog.C`, generates code for accessing a large number of global data items in the shared library `mylib.sl`, and links with `mylib.sl`.
`+L`	Generates source line number information using the format `#line %d` instead of `#%d`. See also the `-Fc` option.
`+m`	Provides maximum compatibility with the USL C++ implementation. (HP C++ provides optimizations and additional functionality that may not be compatible with other C++ implementations.)
`+O`*options*	Refer to the *HP C++ Online Programmer's Guide* for a discussion of all current optimization options.
`+p`	Disallows all anachronistic constructs. Ordinarily, the compiler gives warnings about anachronistic constructs. Using the `+p` option, the compiler does not compile code containing anachronistic constructs. Refer to *The C++ Programming Language* for a list of anachronisms.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`+R`*num*	Allows only the first *num* `register` variables to actually be promoted to the `register` class. Use this option when the register allocator issues an "out of general registers" message. (The default value is 10.) This option is only used in translator mode (that is, with the `+T` option). It is ignored in compiler mode.
`+T`	Requests translator mode. In translator mode your HP C++ source code is translated to C code, then compiled by the HP C compiler, linked and patched.
`+w`	Warns about all questionable constructs, as well as constructs that are almost certainly problems. The HP C++ default is to warn only about constructs that are almost certainly problems. This option also warns you when calls to inline functions cannot be expanded inline.
`+x`*file*	This option is only valid in translator mode. This option 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 can be useful for cross-compilations and for porting the translator.
`+Xd`	(Versions 10.22 and later.) Prevents the default elimination of of duplicate symbolic debug information. The compiler now eliminates duplicate symbolic debug information to reduce the size of object files and executables and to enhance performance. To prevent the elimination of duplicate symbolic debug information, use the `+Xd` option. This option may only be needed if you have `pxdb` or `xdb` problems. For example: `CC +Xd prog.C -g` prevents the the elimination of duplicate symbolic debug information.

Table 3-1. The CC Command Options (continued)

Option	Effect of Specifying the Option
`+Xehdtcount`	(Versions A.10.09 and later.) Use `+Xehdtcount` with the `+eh` option to generate instrumentation that can locate potential run-time exception handling problems. Use this option when the application exhibits run-time range errors or aborts. You may also use this option during development, in case problems arise.
`+z`	Causes the compiler to generate position-independent code (PIC), necessary for building shared libraries. The options `-g`, `-g1`, `-G`, and `-p` are ignored if either `+z` or `+Z` is used. See also the `-b` and `+Z` options. For more information on shared libraries, see "Creating and Using Shared Libraries" in this chapter, and the manual *HP-UX Linker and Libraries Online User Guide*.
`+Z`	This option is the same as the `+z` option except it allows for more imported symbols than `+z` does. In general, use the `+z` option unless you get a linker error message indicating that you should use `+Z`.

Note

The `+e0/e1` options were used in earlier versions of `cfront` to determine when to emit the virtual table for a class. These options are still available but they have no effect in most cases. Currently `cfront` emits the definitions of the virtual table in the compilation unit that contains the definition (not declaration) of the first function in the class that is virtual and not inline. If there is no such function, multiple virtual tables definitions might still be emitted. For example, if you have a class in which all of the virtual functions are inline, then, by default, `cfront` emits a virtual table in every compilation unit that uses this class. In such cases, the `+e0/+e1` options can be used to control when to emit the virtual function table. In other words, the `+e0/+e1` options are useful only when `cfront` cannot determine a unique place to emit the virtual table.

Note

On the HP 9000 Series 700/800, the default is to allow null-pointer dereferencing, so using `-Z` has no effect.

Unsharable executable files generated with the `-N` option cannot be executed with `exec`. For details and system defaults, refer to the description of `ld` in the *HP-UX Reference Manual* .

Any other options not recognizable by `CC` generate a warning to `stderr`. (Options not recognized by `CC` are not passed to `ld`. Use the `-Wl`, *arg* option to pass options to `ld`.)

Environment Variables

This section describes the following environment variables you can use to control the C++ compiler:

- CXXOPTS
- TMPDIR
- CCLIBDIR
- CCROOTDIR

The CXXOPTS Environment Variable

The compiler divides CXXOPTS options into two sets; those which appear before a vertical bar (|), and those options which appear after the vertical bar. Note that the vertical bar must be delimited by whitespace. The first set of options is placed before the command-line parameters to CC; the second set of options is placed after the command-line parameters to CC. If the vertical bar is not present, all options are placed before the command-line parameters.

Note For C++ versions prior to HP-UX 10.*x*, the vertical bar is n t supported and all options assigned to the CXXOPTS environment variable are placed before the command-line parameters.

CXXOPTS provides a convenient way to include frequently used command line options automatically. Just set the environment variable with the options you want and the command line options are automatically included each time you execute the CC command.

Caution Using the CCOPTS environment variable in **translator mode** can cause unexpected side effects because HP C (cc) uses CCOPTS.

For example:

```
CXXOPTS ="-v | -lm"    sh(1) notation
export CXXOPTS
CC -g prog.C
```

is equivalent to:

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

The following example:


```
export CXXOPTS=-v    ksh notation
setenv CXXOPTS -v    csh notation
```

Causes the option `-v` to be passed to the `CC` command each time you execute the `CC` command.

When `CXXOPTS` is set as above, the following two commands are equivalent:

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

The TMPDIR Environment Variable

Another environment variable, `TMPDIR`, allows you to change the location of temporary files that the compiler creates. The directory specified in `TMPDIR` replaces `/tmp` and `/usr/tmp` (`/var/tmp` for HP-UX 10.*x* C++ versions) as the default directory for temporary files. The syntax for `TMPDIR` in `csh` notation is


```
setenv TMPDIR altdir
```

where *altdir* is the name of the alternative directory for temporary files.

The CCLIBDIR and CCROOTDIR Environment Variables

Two additional environment variables that allow HP C++ to reside in alternate directories are provided. `CCLIBDIR` causes the `CC` command to search for libraries in the alternate directory indicated, rather than in their default directories. The `CCROOTDIR` environment variable causes `CC` to invoke all subprocesses from an alternate directory indicated, rather than from their default directories.

The syntax in `csh` notation is:

```
setenv CCLIBDIR altlibdir
setenv CCROOTDIR altdir
```

Pragma Directives

This section describes the pragmas you can use within an HP C++ source file. A pragma has effect from the point where it is included to the end of the compilation unit or until another pragma changes its status. For more information about pragmas, see "Pragma Directive" in Chapter 2.

Optimization Pragmas

For information on optimization pragmas, see "Pragma Directives" in the *HP C++ Online Programmer's Guide*.

Pragmas for Shared Libraries

This section describes a pragma you can use with shared libraries.

Pragma HP_SHLIB_VERSION.

```
#pragma HP_SHLIB_VERSION ["]date["]
```

With the `HP_SHLIB_VERSION` pragma you can create different versions of a routine in a shared library. `HP_SHLIB_VERSION` assigns a version number to a module in a shared library. The version number applies to all global symbols defined in the module's source file.

The *date* argument is of the form *month/year*. The month must be 1 through 12, corresponding to January through December. The *year* can be specified as either the last two digits of the year (`92` for `1992`) or a full year specification (`1992`). Two-digit year codes from `00` through `40` represent the years 2000 through 2040.

This pragma should only be used if incompatible changes are made to a source file. If a version number pragma is not present in a source file, the version number of all symbols defined in the object module defaults to `1/90`. For more information on shared libraries, see the section "Creating and Using Shared Libraries" later in this chapter. Also see the manual *HP-UX Linker and Libraries Online User Guide*.

Pragma COPYRIGHT.

```
#pragma COPYRIGHT "string"
```

`COPYRIGHT` specifies the name to use in the copyright message, and causes the compiler to put the copyright message in the object file. If no date is specified (using `#pragma COPYRIGHT_DATE "string"` as shown below), the current year is used.

For example, assuming the year is 1990, the directive `#pragma COPYRIGHT "Acme Software"` places the following string in the object code:

```
(C) Copyright Acme Software, 1990. All rights reserved. No part of
this program may be photocopied, reproduced, or transmitted without
prior written consent of Acme Software.
```


Pragma COPYRIGHT_DATE.

```
#pragma COPYRIGHT_DATE "string"
```

`COPYRIGHT_DATE` specifies a date string to be used in a copyright notice appearing in an object module.

Pragma LOCALITY.

```
#pragma LOCALITY "string"
```

`LOCALITY` specifies a name to be associated with the code written to a relocatable object module. All code following the `LOCALITY` pragma is associated with the name specified in `string`. The smallest scope of a unique `LOCALITY` pragma is a function. For example, `#pragma locality "mine"` builds the name `$CODE$MINE$`.

Code that is not headed by a `LOCALITY` pragma is associated with the name `$CODE$`.

Pragma VERSIONID.

```
#pragma VERSIONID "string"
```

This pragma specifies a version string to be associated with a particular piece of code. The string is placed into the object file produced when the code is compiled.

System Library and Header Files

This section discusses the two types of libraries provided with HP C++:

- standard HP-UX libraries
- HP C++ run-time libraries

Standard HP-UX Libraries

There are several libraries providing system services that are included with HP-UX. You can access HP-UX standard libraries by using header files that declare interfaces to those libraries. These library routines are documented in the *HP-UX Reference Manual*.

Location of Standard HP-UX Header Files

The standard HP-UX header files are located in `/usr/include`.

To use a system library function, your HP C++ source code must include the preprocessor directive `#include`. For example,

```
#include <filename.h>
```

where `filename.h` is the name of the C++ header file for the library function you want to use. By enclosing `filename.h` in angle brackets, the HP C++ preprocessor looks for that particular header file in a standard location on the system. The HP C++ preprocessor first looks for header files in `/usr/include/CC` (in `/opt/CC/include` for HP-UX 10.*x* C++ versions); if any are not found, it then searches `/usr/include`.

You can use `-I`*dir* options to modify the search path. If the `-I`*dir* option is specified, the HP C++ preprocessor first looks for `#include` files in the directories specified in *dir* before looking into the standard include directories.

Example of Using a Standard Header File

If you want to use the `getenv` function that is in the standard library files `/lib/libc.sl` and `/lib/libc.a` (or `/usr/lib/libc.sl` and `/usr/lib/libc.a` for HP-UX 10.*x* C++ versions), you should specify


```
#include <stdlib.h>
```

because the external declaration of `getenv` is found in the header file `/usr/include/stdlib.h`.

C++ Run-Time Libraries

In addition to standard HP-UX system libraries, HP C++ provides the following C++ run-time libraries:

Stream Library

The stream library includes class libraries for buffering and formatting I/O operations. It consists of several main I/O classes providing the fundamental facility for I/O conversion and buffering. The stream library also provides classes derived from main classes offering extended I/O functionality such as in-memory formatting and file I/O. For more detailed documentation of this library, refer to the *C++ Language System Library Manual*.

Ostream Library

The Ostream library is no longer provided with HP C++. It was provided with version 2.1 for backward compatibility with the AT&T C++ version 1.2 stream I/O library. The newer C++ stream library (available since version 2.0) is mostly upward compatible with the older stream library, but there are a few places where differences may affect programs. These differences are discussed in chapter 3 of the *C++ Language System Library Manual*, under "Converting from Streams to Iostreams."

Task Library

The task library is a multiple threaded, co-routine class library that enables users to simulate, control, and model UNIX system processes in an object-oriented paradigm. This library also encapsulates reusable tasking primitives such as the scheduler, task queue, timer, and interrupt handler. The

task library is useful for simulations or pseudo parallel-processing algorithms. For more detailed documentation of this library, refer to the *C++ Language System Library Manual*.

Complex Library

The complex library implements the data type of complex numbers as a class `complex`. It overloads the standard input, output, arithmetic, assignment, and comparison operations. It also defines the standard exponential, logarithm, power, and square-root functions, as well as the trigonometric functions of sine, cosine, hyperbolic sine, and hyperbolic cosine. For more detailed documentation of this library, refer to the *C++ Language System Library Manual*.

HP Codelibs Library

The HP Codelibs library contains many general-purpose classes you can use in your applications, including:

- strings
- dynamic arrays
- sets
- hash tables
- shared memory management routines
- memory allocation
- lists

The header files for the HP Codelibs library are in the directory `/usr/include/codelibs` (`/opt/CC/include/codelibs` for HP-UX 10.*x* C++ versions). Use `-I/usr/include/codelibs` to direct the compiler to search these header files. For more information about this library, refer to the *Codelibs Library Reference - Version 2.100*.

System Library and Header Files

Standard Components Library

The USL C++ Standard Components library contains many general-purpose classes you can use in your applications, including:

- dynamic arrays
- graphs
- lists
- memory allocation
- sets
- bags
- strings

The header files for the Standard Components library are in the directory `/usr/include/SC` (`/opt/CC/include/SC` for HP-UX 10.*x* C++ versions). Use `-I/usr/include/SC` to direct the compiler to search these header files.

A collection of program development tools for use with the Standard Components library are in `/usr/bin`. These include `hier`, `incl`, `publik`, `dem`, and `g2++comp`.

For more information about the Standard Components, refer to the *USL C++ Standard Components Manual*. To see the online manual pages, first add the directory `/usr/CC/man/SC` (`/opt/CC/share/man` for HP-UX 10.*x* C++ versions) to your `MANPATH` environment variable. Then type man name where name is a particular manual page. For an introduction to Standard Components, type `man SC_intro` and `man SC_tools_intro`.

Locations of Library Files

Table 3-2 lists the files containing the HP C++ run-time libraries.

Different libraries are used depending on whether or not you use exception handling.

Table 3-2. HP C++ Library Files

Library	Default File	Exception Handling File

Stream library	`/usr/lib/libC.a`
	`/usr/lib/libC.sl`
	`/usr/lib/libC.ansi.a`
	`/usr/lib/libC.ansi.sl`
		`/usr/lib/CC/eh/libC.a`
		`/usr/lib/CC/eh/libC.ansi.a`

Task library	`/usr/lib/libtask.a` `/usr/lib/libtask.sl`	Not available.
Complex library	`/usr/lib/libcomplex.a`	`/usr/lib/CC/eh/libcomplex.a`
Demangling library	`/usr/lib/libdemangle.a`	Not available.
Codelibs library	`/usr/lib/libcodelibs.a` `/usr/lib/libcodelibs.sl`	`/usr/lib/CC/eh/libcodelibs.a`

Standard Components library	`/usr/lib/lib++.a`
	`/usr/lib/libGA.a`
	`/usr/lib/libGraph.a`
	`/usr/lib/libfs.a`
	`/usr/lib/libg2++.a`
	`/usr/lib/incl2`
	`/usr/lib/hier2`
	`/usr/lib/publik2`
		`/usr/lib/CC/eh/lib++.a`
		`/usr/lib/CC/eh/libGA.a`
		`/usr/lib/CC/eh/libGraph.a`
		`/usr/lib/CC/eh/libfs.a`
		`/usr/lib/CC/eh/libg2++.a`
		`/usr/lib/CC/eh/incl2`
		`/usr/lib/CC/eh/hier2`
		`/usr/lib/CC/eh/publik2`

Note HP-UX 10.*x* C++ versions store library files in `/opt/CC/lib`.

Support for Multi-Threaded Applications

For HP-UX 10.*x* C++ versions, the HP C++ run-time environment supports multi-threaded applications. The following HP C++ libraries are thread-safe:

- libC.ansi.sl and libC.ansi.a
- libC.sl and libC.a
- libcxx.a

- libcomplex.a

There are no interface changes to the functions in these libraries nor in the standard include files in `/usr/include` and `/opt/CC/include/CC`. However, there are new reentrant versions of the functions chr_r, str_r, form_r, hex_r, dec_r, and oct_r. In order to link with these functions, you must include the stream.h header file in your source file and add the -D_THREAD_SAFE compile time flag to your compilation line.

Also, in order to pick the thread safe version of the I/O routines, cout, cin, and cerr, you must include iostream.h in your source files and add the -D_THREAD_SAFE compile time flag to your compilation line. To guarantee that your I/O results from one thread are not intermingled with I/O results from other threads, you should protect your I/O statement with locks. For example:

```
// create a mutex and initialize it
pthread_mutex_t the_mutex;
pthread_mutex_init(&the_mutex, pthread_mutexattr_default);

pthread_mutex_lock(&the_mutex);
  cout << "something" ... ;
pthread_mutex_unlock(&_the_mutex);
```

There are no new compiler options for compiling multi-threaded programs.

C++ Library Header Files

HP C++ includes the following header files for interface to C++ run-time libraries:

- `complex.h` — implementation of complex numbers class
- `generic.h` — error handling and string concatenation macros
- `iostream.h` — I/O streams classes `ios`, `istream`, `ostream`, and `streambuf`
- `fstream.h` — I/O streams specialized for files
- `strstream.h` — `Streambuf` specialized to arrays
- `iomanip.h` — predefined manipulators and macros

- `stdiostream.h` — specialized `streams` and `streambufs` for interaction with `stdio`
- `stream.h` — includes `iostream.h`, `fstream.h`, `stdiostream.h` and `iomanip.h` for compatibility with AT&T USL C++ version 1.2
- `vector.h` — macros for class declaration and constructor definition for vectors
- `task.h` — implementation of task class
- `dem.h` — routines for demangling encoded C++ names (not compatible with previous version)
- `eh.h` — exception handling routines
- `new.h` — dynamic memory routines `new` and `set_new_handler`

For more detailed documentation on these headers, refer to *C++ Language System Library Manual*. For information on the header files for the HP Codelibs library, refer to the *Codelibs Library Reference - Version 2.100*.

For information on the header files for the Standard Components library, refer to the *USL C++ Standard Components Manual*.

Location of C++ Header Files

The above header files are located in the directory `/usr/include/CC`. The header files for the HP Codelibs library are in the directory `/usr/include/codelibs`. (In `/opt/CC/include/codelibs` for HP-UX 10.*x* C++ versions.)

Use `-I/usr/include/codelibs` to direct the compiler to search these header files. The header files for the Standard Components library are in the directory `/usr/include/SC`. (In `/opt/CC/include/SC` for HP-UX 10.*x* C++ versions.) Use `-I/usr/include/SC` to direct the compiler to search these header files.

Example of Using a C++ Header File

If, for example, you want to use complex numbers in your application, you must specify the following:

```
#include <complex.h>
```

Linking to C++ Libraries

You can compile and link any C++ modules to one or more libraries. HP C++ automatically links

`/usr/lib/libC.sl`

(the HP C++ run-time library, including the stream library) and

`/lib/libc.sl`

(the HP-UX system library) with a C++ program.

The ANSI C versions of the C++ run-time library are also included, `libC.ansi.sl` and `libC.ansi.a`. If you have compiled with the `+a1` option, you must also pass `+a1` to the `CC` command when linking to make sure the linker uses these libraries.

If you want archive libraries instead of shared libraries, use the `-a,archive` linker option. (See the section "Linking Archive or Shared Libraries" later in this chapter for more information.)

You can specify other libraries using the `-l` option. For example, in order to use the complex library, you must specify `-lcomplex`:

```
CC complex_appl.C -lcomplex
```

Your C++ run-time library may require that additional HP-UX standard libraries be specified. For example, the complex library uses the HP-UX math library for mathematical functions. So, for example, you might need to specify `-lm` for the math library:

```
CC complex_appl.C -lcomplex -lm
```

Creating and Using Shared Libraries

This section provides information about shared libraries that is specific to HP C++. For additional information about creating and using shared libraries, see the manual *HP-UX Linker and Libraries Online User Guide*. For information on using the options to the `CC` command, see Table 3-1 in this chapter.

Compiling for Shared Libraries

To create a C++ shared library, you must first compile your C++ source with either the `+z` or `+Z` option. These options create object files containing position-independent code (PIC).

Creating a Shared Library

To create a shared library from one or more object files, use the `-b` option at link time. (The object files must have been compiled with `+z` or `+Z`.) The `-b` option creates a shared library rather than an executable file.

Note You must use the `CC` command to create a C++ shared library. This is because the `CC` command ensures that any static constructors and destructors in the shared library are executed at the appropriate times.

Using a Shared Library

To use a shared library, you simply include the name of the library on the `CC` command line as you would with an archive library, or use the `-l` option, as with other libraries. The linker links the shared library to the executable file it creates. Once you create an executable file that uses a shared library, you must not move the shared library or the dynamic loader (`dld.sl`(5)) will not be able to find it.

Note You must use the `CC` command to link any program that uses a C++ shared library. This is because the `CC` command ensures

that any static constructors and destructors in the shared library are executed at the appropriate times.

Example

The following command compiles the two files `Strings.C` and `Arrays.C` and creates the two object files `Strings.o` and `Arrays.o`. These object files contain position-independent code (PIC):

```
CC -c +z Strings.C Arrays.C
```

The following command builds a shared library named `libshape.sl` from the object files `Strings.o` and `Arrays.o`:

```
CC -b -o libshape.sl Strings.o Arrays.o
```

The following command compiles a program, `draw_shapes.C`, that uses the shared library, `libshape.sl`:

```
CC draw_shapes.C libshape.sl
```

Linking Archive or Shared Libraries

If both an archive and shared version of a particular library reside in the same directory, the linker links in the shared version by default. You can override this behavior with the `-a` linker option. This option tells the linker which type of library to use. The `-a` option is positional and applies to all subsequent libraries specified with the `-l` option until the end of the command line or until the next `-a` option is encountered.

The syntax of this option when used with `CC` is:

```
-Wl,-a,{ archive }
       { shared  }
       { default }
```

The different meanings of this option are:

`-Wl,-a,archive` Select archive libraries. If the archive library does not exist, the linker generates a warning message and does not create the output file.

`-Wl,-a,shared`	Select shared libraries. If the shared library does not exist, the linker generates a warning message and does not create the output file.
`-Wl,-a,default`	Select the shared library if it exists; otherwise, select the archive library.

The following example directs the linker to use the archive version of the library `libshape`, followed by standard shared libraries if they exist; otherwise select archive versions.

```
CC box.o sphere.o -Wl,-a,archive -lshape -Wl,-a,default
```

Updating a Shared Library

The `CC` command cannot replace or delete object modules in a shared library. To update a C++ shared library, you must recreate the library with *all* the object files you want the library to include. If, for example, a module in an existing shared library requires a fix, simply recompile the fixed module with the `+z` or `+Z` option, then recreate the shared library with the `-b` option. Any programs that use this library will now be using the new versions of the routines. That is, you do not have to relink any programs that use this shared library because they are attached at run time.

Forcing the Export of Symbols in main

By default, the linker exports from a program only those symbols that were imported by a shared library. For example, if an executable's shared libraries do *not* reference the program's `main` routine, the linker does *not* include the `main` symbol in the `a.out` file's export list. Normally, this is a problem only when a program explicitly calls shared library management routines. (See "Routines You Can Use to Manage C++ Shared Libraries" later in this chapter.) To make the linker export *all* symbols from a program, use the `-Wl,-E` option which passes the `-E` option to the linker.

Binding Times

Because shared library routines and data are not actually contained in the `a.out` file, the dynamic loader must **attach** the routines and data to the program at run time. To accelerate program startup time, routines in a shared library are not bound until referenced. (Data items are always bound at program startup.) This deferred binding distributes the overhead of binding across the total execution time of the program and is especially helpful for programs that contain many references that are not likely to be executed.

Forcing Immediate Binding

You can force immediate binding, which forces all routines and data to be bound at startup time. With immediate binding, the overhead of binding occurs only at program startup time, rather than across the program's execution. Immediate binding also detects unresolved symbols at startup time, rather than during program execution. Another use of immediate binding is to get better interactive performance, if you don't mind program startup taking longer. To force immediate binding, use the option `-Wl,-B,immediate`. For example,

```
CC -Wl,-B,immediate draw_shapes.o -lshape
```

To get the default binding, use `-Wl,B,deferred`. For more information, see *HP-UX Linker and Libraries Online User Guide*.

Side Effects of C++ Shared Libraries

When you use C++ shared libraries, all constructors and destructors of nonlocal static objects in the library execute. This is different from C++ archive libraries where only the constructors and destructors in object files *you actually use* are executed.

Routines You Can Use to Manage C++ Shared Libraries

You can call any of several routines to explicitly load and unload shared libraries, and get information about shared libraries. Refer to *HP-UX Linker and Libraries Online User Guide* for information about these routines.

HP C++ provides the following additional routines for managing C++ shared libraries:

`cxxshl_load()`	Explicitly loads a shared library and executes any constructors for nonlocal static objects if they exist. This routine is identical to the general `shl_load()` routine except that it also executes appropriate constructors. See *HP-UX Linker and Libraries Online User Guide* for information about `shl_load()`, including syntax. `cxxshl_load()` can be used on non-C++ shared libraries and can be called from other languages.
`cxxshl_unload()`	Executes the destructors for any constructed nonlocal static objects and unloads the shared library. This routine is identical to the general `shl_load()` routine except that it also executes appropriate destructors. See *HP-UX Linker and Libraries Online User Guide* for information about `shl_load()`, including syntax. `cxxshl_unload()` can be used on non-C++ shared libraries and can be called from other languages.

When you use either of these routines, be sure to compile with the `-ldld` option to link them in.

Shared Library Header files

The C++ shared library management routines (`cxxshl_load()` and `cxxshl_unload()`) use special data types and constants defined in the header file `/usr/include/CC/cxxdl.h` (`/opt/CC/include/cxxdl.h` for HP-UX 10.*x* C++ versions). When using these functions from a C++ program be sure to include `cxxdl.h`:

```
#include <cxxdl.h>
```

If an error occurs when calling shared library management routines, the system error variable, `errno`, is set to an appropriate error value. Constants are defined for these error values in `/usr/include/errno.h` (see *errno*(2)). Thus, if a program checks for these values, it must include `errno.h`:

```
#include <errno.h>
```

Version Control in Shared Libraries

You can create different versions of a routine in a shared library with the `HP_SHLIB_VERSION` pragma. `HP_SHLIB_VERSION` assigns a version number to a module in a shared library. The version number applies to all global symbols defined in the module's source file. The syntax of this pragma is:

```
#pragma HP_SHLIB_VERSION ["]date["]
```

The *date* argument is of the form *month/year*. The month must be 1 through 12, corresponding to January through December. The *year* can be specified as either the last two digits of the year (`92` for `1992`) or a full year specification (`1992`). Two-digit year codes from `00` through `40` represent the years 2000 through 2040.

This pragma should only be used if incompatible changes are made to a source file. If a version number pragma is not present in a source file, the version number of all symbols defined in the object module defaults to `1/90`. For more information about version control in shared libraries, see *HP-UX Linker and Libraries Online User Guide*.

Adding New Versions to a Shared Library

To rebuild a shared library with new versions of some of the object files, use the `CC` command and the `-b` option with the old object files and the newly compiled object files. The new source files should use the `HP_SHLIB_VERSION` pragma.

For example, suppose the source file `box.C` has been compiled into the shared library `libshape.sl`. Further suppose you want to add new functionality to functions in `box.C`, making them incompatible with existing programs that call `libshape.sl`. Before making the changes, make a copy of the existing `box.C` and name it `oldbox.C`. Then change the routines in `box.C`, using the version pragma specifying the current month and year. The following illustrates these steps:

```
cp box.C oldbox.C     Save the old source.
mv box.o oldbox.o     Save the old object file.
   ⋮                  Change box.C to create the new version.

#pragma HP_SHLIB_VERSION "9/92"  // Date is September 1992.
```

```
// This is a new version of the box class, in box.C.
            box::box() {...}
```

To update the shared library, `libshape.sl`, to include the new `box.C` routines, compile `box.C` and rebuild the library with the new `box.o` and `oldbox.o`:

```
CC -c +z box.C
CC -b -o libshape.sl oldbox.o sphere.o box.o
```

Thereafter, any programs linked with `libshape.sl` use the new versions of the `box.C` routines. Programs linked with the old version still use the old versions.

Distributing HP C++ Libraries, Object Files, and Executable Files

If you write applications in HP C++ and distribute them to your customers, see "Distributing Your C++ Application" in "Compiling and Executing HP C++ Programs" in the *HP C++ Online Programmer's Guide*.

Executing HP C++ Programs

After a program is successfully linked, it is in executable form.

To execute a program, enter the executable file name (either `a.out` or the file name you used following the `-o` option). For example, to execute an object file named `my_executable`, enter:

```
my_executable
```

The operating system searches for a file named `my_executable` according to its usual search rules, calls the loader utility, and then executes the program.

Redirecting stdin and stdout

By default, standard input (stdin) and output (stdout) for the program are assigned to the keyboard and display, respectively. You can direct standard input and output by using the shell's redirection notation. For example, to redirect standard input when you invoke `my_executable`, enter:

```
my_executable  < input_data
```

The `<` character reassigns standard input to the file `input_data`. You can redirect standard output in a similar fashion. For example,

```
my_executable > results
```

This command uses the character `>` to redirect standard output for `my_executable` to the file named `results`.

An Extensive Example

This section describes one model for designing a typical kind of C++ program. There are many ways to design a program. The method described here illustrates an object-oriented approach that uses data hiding.

The discussion is organized according to the following topics:

- data hiding using files as modules
- linking
- an example based on a lending library

Data Hiding Using Files as Modules

Most programs are made up of several separately compiled units, usually files. The term *module* refers to a file containing a variable or function declaration, a function definition, or several of these or similar items logically grouped together. Thus, a program usually consists of several modules.

A C++ *service* consists of the following:

- The declaration of all the objects the service provides. This is called the *interface*.
- The operations that the service performs with its objects. This is called the *implementation* of those objects.

The interface is usually in one or more header files, or `.h` files. The implementation is usually in one or more `.C` or `.c` files associated with the corresponding `.h` files. Code in an application using the service is sometimes called a *client* of the service. Client source code is usually in a `.C` or a `.c` file.

Suppose, for example, a simple lending library service is organized into two modules, `library_ex.h` and `library_ex.C`.

The source file for this program resides in the directory `/usr/contrib/CC/Examples` (or `/opt/CC/contrib/Examples/library_ex` for HP-UX 10.*x* C++ versions.) The interface module, `library_ex.h`, contains the declarations of the objects in the service. Perhaps these would be class types named `library`, `book`, `borrower`, and `transaction`. The implementation module, `library.C`, contains the function definitions for the objects in the interface. Examples of these might be function definitions for

`library::display_books ()` and `library::add_book(book*)`. A client of the library service could then consist of code in a `.C` file such as `use_library.C`. The sample program at the end of this chapter (example 3-1) is organized in just this way.

This type of organization uses *data hiding*, since it allows you to make available to clients of the service only the names they need to know. You can hide information that a client need not know in the `.C` files, or, if necessary, keep the implementation in object file format (`.o` files) only.

This type of service also provides considerable flexibility. An implementation can consist of one or more `.C` files, and you can provide several different interfaces in the form of `.h` files.

The next section describes how the separate modules of the service can be linked together.

Linking

Just like a program consisting of a single source file, a program consisting of many separately compiled parts must be consistent in its use of names and types.

For instance, a name that is not local to a function or a class must refer to the same type, value, function, or object in every separately compiled part of a program. That is, there can only be one nonlocal type, value, function, or object in a program with that name.

An object may be declared many times, but it must be defined exactly once. Also, the types must agree exactly in all the declarations of the object. Constants, inline functions, and type definitions can be defined as many times as necessary, but they must be defined identically everywhere in the program.

The best way to ensure that the declarations in separate modules are consistent is to follow these steps:

1. Use a `#include` in each of your `.C` implementation files and `.C` client files.
2. Compile each `.C` or `.c` file with `CC` using the `-c` option. This step creates an object file with an `.o` suffix.
3. Link the object files created in step 2 using `CC`. This step creates an executable file.

In example 3-1 of a library service, the `use_library.C` and `library_ex.C` files each contain the following line:

```
#include "library_ex.h"
```

You could generate an object file from `library_ex.C` using the following command:

```
CC -c library_ex.C
```

Similarly, you generate an object file from `use_library.C` using the following command line:

```
CC -c use_library.C
```

Finally, you link the object files to create an executable file named `a.out`, using the following command:

```
CC use_library.o library_ex.o
```

The Lending Library

This section presents a simple example of a C++ service. The example is not intended to be a realistic application, but it illustrates the organization and concepts that have been discussed in this section.

The service example is a lending library. Its principal objects correspond to the books in the library's collection (`book`) and people who are enrolled to borrow books (`borrower`). The service includes an interaction object (`transaction`), which associates a particular book with a particular borrower, and an object that contains `book`, `borrower`, and `transaction` objects (`library`). There is an abstract data type (`list`) from which all the other classes are derived, making it easier to handle lists of the various types of objects.

The interface for the service is in the file `library_ex.h`, which lists the declarations for the `book`, `borrower`, `transaction`, `library`, and `list` classes. The implementation for the service is in the file `library_ex.C`, which lists the definitions for the `book`, `borrower`, `transaction`, `library`, and `list` classes. A client application program is listed in `use_library.C`.

The source file for this program resides in the directory `/usr/contrib/CC/Examples` (or `/opt/CC/contrib/Examples/library_ex` for for HP-UX 10.*x* C++ versions.)

To use the lending library service, put your source files in the same directory and follow the steps described above.

To run the executable file, enter:

```
a.out
```

The rest of this chapter consists of Example 3-1, which shows the three files discussed above: `library_ex.h`, `library_ex.C`, and `use_library.C`.

```
//------------------------------------------------------------------------
// library_ex.h  -- the interface for the lending library service
//------------------------------------------------------------------------
// This module is included in the library_ex.C module.
// Some functions declared in this file are defined in the
// library_ex.C module.  You must link the library_ex.C and
// use_library.C modules to create an executable file.
// The main() function is in the use_library.C module.
//------------------------------------------------------------------------
#include <stream.h>
//------------------------------------------------------------------------
class list                      // list is an abstract class
{
private:
     list* link ;               // the only data member is a
                                // pointer to a list object
public:
     list()
      { link = 0; }             // constructor
     list* add(list* p )
      { p->link = this; return p; }
                                // this inline function adds a new
                                // item as the first one on the list
      list* next()
      { return this->link; }    // this inline function returns the
                                // next link on the list
     virtual void display_item()
      { cout << "none yet"; }   // this virtual function is redefined for
                                // book, borrower, and transaction classes
     void display();            // this function calls display_item()
};
//------------------------------------------------------------------------
class book:public list         // book is derived from list
{
private:
    char* title;                // data members are strings for
    char* author;               // the book's title and author
public:
    book(char* t,char* a);      // constructor for a book
    book* add_book(book* b)     // adds a book to the list
         { return (book*)(this->add(b)); }
    void display_item()         // shows a book's title and author
         { cout << " " << title << " by " << author;}
};
//------------------------------------------------------------------------
```

Example 3-1. A C++ Service: library_ex.h, library_ex.C, and use_library.C

```
//-----------------------------------------------------------------------
class borrower:public list     // borrower is derived from list
{
private:
    char* last_name;           // data members are strings for
    char* first_name;          // name and address of borrower
    char* address;
public:
    borrower(char* l, char* f, char* a); // constructor for borrower
    borrower* add_borrower(borrower* b)
         { return (borrower*)(this->add(b)); }
                               // adds a borrower to the list
    void display_item()        // shows a borrower's name and address
          {cout << " " << first_name << " " << last_name
                << " of " << address;}
};
//-----------------------------------------------------------------------
class transaction:public list // transaction is derived from list; it
                              // is an interaction object that creates
                              // an association between two objects
{
private:
    borrower* person;         // person who borrowed the book
    book* a_book;             // book that was borrowed public:
public:
    transaction(borrower* p, book* b);  //constructor
    transaction* add_transaction(transaction* t)
         { return (transaction*)(this->add(t)); }
                              // adds a transaction to the list
    void display_item()       // shows the book on loan to a borrower
         { person->display_item(); cout << " borrowed" ;
           a_book->display_item(); }
};
//-----------------------------------------------------------------------
```

Example 3-1. A C++ Service: library_ex.h, library_ex.C, and use_library.C (continued)

```
//-----------------------------------------------------------------------
class library  // a library object contains linked lists of book,
               // borrower, and transaction objects
{
private:
     book* books;              // these are pointers to the first
     borrower* borrowers;      // object of each type on the list
     transaction* transactions;
public:
     library()
         { books = 0; borrowers = 0; transactions =0; }
                               // initialize the lists to null pointers
     void add_book (book* b)
         { if (books == 0) books = b; else books=books->add_book(b);}
                               // adds a book to the library's collection
     void add_borrower( borrower* b)
         { if (borrowers == 0) borrowers = b;
           else borrowers=borrowers->add_borrower(b); }
                               // enrolls a borrower as a library patron
     void add_transaction (transaction* t)
         { if (transactions == 0) transactions = t;
           else transactions=transactions->add_transaction(t); }
                               // records a book borrowed by a borrower
     void  display_books()
         { books->display(); }
                               // show all books in the library
     void  display_borrowers()
         { borrowers->display(); }
                               // show all borrowers currently enrolled
     void  display_transactions()
         { transactions->display(); }
                               // show all books currently borrowed
};
//-----------------------------------------------------------------------
```

Example 3-1. A C++ Service: library_ex.h, library_ex.C, and use_library.C (continued)

```
//--------------------------------------------------------------------
// library_ex.C -- the implementation for the interface in
//                 the library_ex.h header file
//--------------------------------------------------------------------
// This module can be linked with a client file, for example, the
// use_library.C module, to create an executable file. The
// main() function should be in the client file.
//--------------------------------------------------------------------
#include "library_ex.h"
#include <string.h>

//-----------------display a list ---------------------------------
void list::display()
{
 list* root = this;            // start at the beginning of the list
 if (root == 0)
    cout << "\nnone right now "; // check for empty list
else
    while (root!=0)           // walk through list
    {
     root->display_item();    // calls a virtual function to display
                              // each item using the member
                              // function that corresponds to the
                              // type of this object
     cout << "\n";
     root=root->next();
    }
}
//-----------------construct a book ---------------------------------
book::book(char* t,char* a)
{
  title = new char [strlen(t) +1];  //allocate memory for title
  strcpy(title,t);                  //copy title
  author = new char [strlen(a) +1]; //allocate memory for author
  strcpy(author,a);                 //copy author
}
//--------------------------------------------------------------------
```

Example 3-1. A C++ Service: library_ex.h, library_ex.C, and use_library.C (continued)

```
//-----------------construct a borrower---------------------------
borrower::borrower(char* f, char* l, char* a)
{
  first_name = new char [strlen(f) +1];
                                    //allocate memory for first name
  strcpy(first_name,f);             //copy first name
  last_name = new char [strlen(l) +1];
                                    //allocate memory for last name
  strcpy(last_name,l);              //copy last name
  address = new char [strlen(a) +1];  //allocate memory for address
  strcpy(address,a);                //copy address
}
//---------------construct a transaction ----------------------------
transaction::transaction(borrower* p,book* b)
{
  person = p;
  a_book=b;
}


//------------------------------------------------------------------------
// use_library.C --things you can do with the library service
//------------------------------------------------------------------------
// This program demonstrates the use of a library service. It must
// be linked with library_ex.C to create an executable file.
//------------------------------------------------------------------------
#include "library_ex.h"
main()
{
// Create a library object named the_library
  library* the_library = new library();

// Create some borrowers and add them to the_library
  borrower* me = new borrower ("Tech","Writer","HP");
  borrower* mary = new borrower ("Mary","Hartman","TVLand");
  borrower* mickey = new borrower ("Mickey","Mouse","DisneyLand");
  the_library->add_borrower(me);
  the_library->add_borrower(mary);
  the_library->add_borrower(mickey);
```

Example 3-1. A C++ Service: library_ex.h, library_ex.C, and use_library.C (continued)

```
// Create a few books for the_library
  book* one = new book ("The C Programming Language", "Kernighan and Ritchie");
  book* two = new book ("The French Chef","Julia Child");
  book* three = new book ("HP C++","Tech Writer");
  the_library->add_book(one);
  the_library->add_book(two);
  the_library->add_book(three);

// Create a few transactions for the_library
  transaction* first = new transaction (me,two);
  transaction* second = new transaction (mary,one);
  the_library->add_transaction(first);
  the_library->add_transaction(second);

// Interact with a user
  cout << "\n\nWelcome to the library!  ";
  char  answer = 'Y';
  while ((answer != 'Q') && (answer != 'q'))
{
  cout << "\n\nYou can do the following: \n";
  cout << "\nA   Display the Library Collection ";
  cout << "\nB   Display a List of Borrowers in the Library";
  cout << "\nC   Display the Books on Loan ";
  cout << "\nD   Add a Book to the Library Collection";
  cout << "\nE   Enroll a Borrower ";
  cout << "\nF   Borrow a Book ";
  cout << "\n\nWhat would you like to do?";
  cout << "\nPress A, B, C, D, E, F, or Q to quit.";
  cin >> answer;
  char c;
  cin.get(c);          // read newline
  char string1[80], string2[80], name1[80], name2[80], name3[80];
  switch (answer)
     {
      case 'A': case 'a':  // display the library collection
           {cout << "\nHere's a list of the books in the library :\n";
           the_library->display_books();
           break;}
      case 'B': case 'b': // display a list of borrowers in the library
           {cout << "\nHere's a list of the borrowers:\n";
           the_library->display_borrowers();
           break;}
      case 'C': case 'c': // display the books on loan
           {cout << "\nHere's a list of the books that are out:\n";
           the_library->display_transactions();
           break;}
```

Example 3-1. A C++ Service: library_ex.h, library_ex.C, and use_library.C (continued)

```
      case 'D': case 'd': // Add a book to the library collection
           {cout << "\nWhat is the title of the book to be added ? ";
           cin.get(string1,80); // read characters up to newline
           cin.get(c);          // read newline character
           cout << "\nAnd what is the author's name? ";
           cin.get(string2,80);
           cin.get(c);
           book* newbook = new book (string1,string2);
           the_library->add_book(newbook);
           break;}
      case 'E': case 'e': // Enroll a borrower
           {cout << "\nPlease enter the first name of the borrower-- ";
           cin >> name1;
           cout << "And the last name? ";
           cin >> name2;
           cout << "And where is " << name1 << " "
                << name2 << " from? ";
           cin >> name3;
           borrower* newborrower = new borrower (name1,name2,name3);
           the_library->add_borrower(newborrower);
           break;}
      case 'F': case 'f': // Borrow a book
           {cout << "\nBorrowing a book";
           cout << "\nWhat is the title of the book?  ";
           cin.get(string1,80);
           cin.get(c);
           cout << "\nAnd what is the author's name?  ";
           cin.get(string2,80);
           cin.get(c);
           book* B = new book (string1,string2);
           cout << "\nPlease enter the first name of the borrower-- ";
           cin >> name1;
           cout << "\nAnd the last name? ";
           cin >> name2;
           cout << "\nAnd where is " << name1 << " "
                << name2 << " from? ";
           cin >> name3;
           borrower* C = new borrower (name1,name2,name3);
           transaction* D = new transaction (C,B);
           the_library->add_transaction(D);
           break;}
     }
}
}
//---------------------------------------------------------------------
```

Example 3-1. A C++ Service: library_ex.h, library_ex.C, and use_library.C (continued)

4

Optimizing HP C++ Programs

HP C++ programs can be optimized for improved efficiency. The HP C++ compiler provides levels of optimization options to the `CC` command, and pragmas to control optimization. For detailed information on optimization, refer to the *HP C++ Online Programmer's Guide*.

5

Inter-Language Communication

This chapter provides guidelines for linking HP C++ modules with modules written in HP C, HP Pascal, and HP FORTRAN 77 on HP 9000 Series 700/800 systems.

Introduction

A 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 since C++ is essentially a superset of C. Mixing C++ modules with modules in languages other than C is more complicated.

When creating an executable file from a group of programs of mixed languages, one of them being C++, you need to be aware of the following:

- In general, the overall control of the program must be written in C++. In other words, the `main()` function should appear in a C++ module.
- You must pay attention to case-sensitivity conventions for function names in the different languages.
- You must make sure that the data types in the different languages correspond. Do not mismatch data types for parameters and return values.
- Storage layouts for aggregates differ between 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.

Note HP C++ classes are not accessible to non-C++ routines.

Data Compatibility between C and C++

Since C++ is a superset of C, many of the data types are identical. Both languages have the identical primitive types `char`, `short`, `int`, `long`, `float`, and `double`. ANSI C and C++ also support a `long double` type (and include a language extension for a `long long` type for versions 10.22 and later.)

Pointers, structs, and unions that can be declared in C are also compatible. Arrays composed of any of the above types are compatible.

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

- multiple visibility of members (that is, having both `private` and `public` data members in a class)
- inheritance, either single or multiple
- virtual functions

It is the use of these features, as opposed to whether the `class` keyword is used rather than `struct`, that introduces incompatibilities with C structs.

Calling HP C from HP C++

Since C++ is essentially a superset of C, calling between C and C++ is a normal operation. You should, however, be aware of the following:

- You must use the `extern "C"` linkage specification to declare the C functions.
- Because of function prototypes, C++ has argument-widening rules that are different from C's rules.
- The overall control of the program should be written in C++.

The following sections discuss these issues.

Using the extern "C" Linkage Specification

To handle overloaded function names the HP C++ compiler generates new, unique names for all functions declared in a C++ program. To do so, the compiler uses a function-name encoding scheme that is implementation dependent. A *linkage directive* tells the compiler to inhibit this default encoding of a function name for a particular function.

If you want 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. In other words, you must tell the compiler not to generate a new name for the function. If you don't turn off the usual encoding scheme, the function name declared in your C++ program won't match the function name in your C module defining the function. If the names don't match, the linker cannot resolve them. To avoid these linkage problems, use a linkage directive when you declare the C function in the C++ program.

Calling HP C from HP C++

All HP C++ linkage directives must have either of the following formats:

```
extern "C" function_declaration

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

For instance, the following declarations are equivalent:

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

and

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

You can also use a linkage directive with all the functions in a file, as shown in the following example. This is useful if you wish to use C library functions in a C++ program.

```
extern "C"
{
 #include <string.h>
}
```

Although the string literal following the `extern` keyword in a linkage directive is implementation dependent, all implementations must support C and C++ string literals. Refer to "Linkage Specifications" in *The C++ Programming Language*, and to "Type-Safe Linkage for C++" in the *C++ Language System Selected Readings* for more details about linkage specifications.

Differences in Argument Passing Conventions

By default, the HP C++ compiler in translator mode does not generate function prototypes in the C code it creates. As a result HP C 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, since the arguments are consistently handled within the program. However, if your C++ code calls functions written in C, you should make sure that the called C functions do not use function prototypes that suppress argument widening. If they do, your C++ code will be passing "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`

(or `libC.sl`), which is named `libC.ansi.a` (or `libC.ansi.sl`).

Compiler mode is compatible with translator mode even though no C code is generated. In compiler mode, when you use `+a0`, the default, parameters of type `float` are promoted to type `double`. When you use `+a1`, `float` parameters are not promoted, but are passed as type `float`.

The main() Function

When mixing C++ modules with C modules, the overall control of the program must be written in C++, with two exceptions. In other words, the `main()` function should appear in some C++ module, rather than in a C module. The exceptions are programs without any global class objects containing constructors or destructors and programs without static objects.

Example 5-1 shows a C++ program, `calling_c.C`, that contains a `main()` function. In this example the C++ program calls a C function, `get_name()`. Example 5-2 shows the C function.

```
//*********************************************************
// 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 <iostream.h>
#include "string.h"
//*********************************************************
// declare the external C module
extern "C" char* get_name();
class account
{
private:
    char* name;        // owner of the account
protected:
    double balance;    // amount of money in the account
public:
    account(char* c)   // constructor
        { 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());
  // send a message to my_checking_account to display itself
  my_checking_acct->display();
}
```

Example 5-1. A C++ Program Calling a C Function

The following is example 5-2 showing the module `get_name.c`. This function is called by the C++ program in example 5-1.

```
/*****************************************************/
/* 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;
}
/*****************************************************/
```

Example 5-2. A C Function Called by a C++ Program

Here's a sample run of the executable file that results when you link the object modules generated by compiling `calling_c.C` and `get_name.c`:

```
Enter the name: Janice
Janice has a balance of 0
```

Calling HP C++ from HP C

Examples 5-3 and 5-4 show an example of calling HP C++ from HP C. Example 5-3 is the C++ module and example 5-4 is the C program. These examples illustrate the following points:

- To prevent a function name from being mangled, the function definition and all declarations used by the C++ code must use `extern "C"`.
- The C programmer must generate a call to function `_main` as the first executable statement in `main()`. Object libraries require this as `_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 non-member 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 user to call these interface routines to create such objects before using them and to destroy them afterwards.
- The C user should not try to 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.
- This example also illustrates reference parameters in the interface routine to the constructor.

```
//************************************************
//  C++ module that manipulates object obj.       *
//************************************************
#include <iostream.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";
}
```

Example 5-3. A C++ Module Called by a C Program

Calling HP C++ from HP C: An Example

Example 5-4 is a C program that calls the C++ module in example 5-3 to manipulate the object:

```
/************************************************/
/* C program to demonstrate an interface to the    */
/* C++ module.  Note that the application needs    */
/* to be linked with the CC driver.                */
/************************************************/
typedef struct obj* obj_ptr;

main () {
     /* C++ object. Notice that 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 that static objects 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. Notice 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);
}
```

Example 5-4. A C Program Calling a C++ Module

To compile the programs in examples 5-3 and 5-4, enter the following commands:

```
cc -c cfilename.c
CC -c C++filename.C
CC -o executable cfilename.o C++filename.o
```

Caution During the linking phase, the `CC` driver program performs several functions to support the C++ class mechanism. Linking programs that use classes with the C compiler driver `cc` leads to unpredictable results at run time.

Calling HP Pascal and HP FORTRAN 77 from HP C++

This section covers the following topics related to calling HP Pascal and HP FORTRAN 77 from HP C++:

- the `main()` function
- function naming conventions
- using reference variables to pass arguments
- using `extern "C"` linkage
- strings
- arrays
- definition of TRUE and FALSE
- files

As is the case with calling HP C from HP C++, you must link your application using the C++ driver, `CC`.

The main() Function

In general, when mixing C++ modules with modules in HP Pascal and HP FORTRAN 77, the overall control of the program must be written in C++. In other words, the `main()` function must appear in some C++ module.

Note If you wish to have a `main()` function in a module other than a C++ module, you can add a call to `_main()` as the first non-declarative statement in the module. However, if you use this method, your code is not portable.

Function Naming Conventions

When calling a HP Pascal or HP FORTRAN 77 function from C++, you must keep in mind the differences between the way the languages handle case sensitivity. HP FORTRAN 77 and HP Pascal are not case sensitive, while the C++ compiling system and the underlying C compiler are case sensitive. Therefore, all C++ global names accessed by FORTRAN 77 or Pascal routines must be lowercase. All FORTRAN 77 and Pascal external names are downshifted by default.

Using Reference Variables to Pass Arguments

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 HP Pascal or HP FORTRAN 77 functions from HP C++, you need to ensure that the caller and called functions use the same method of argument passing for each individual argument. Furthermore, when calling external functions in HP Pascal or HP FORTRAN 77, you must know the calling convention for the order of arguments.

It is not recommended that you pass structures or classes to HP FORTRAN 77 or HP Pascal. For maximum compatibility and portability, only simple data types should be passed to routines. All HP C++ parameters are passed by value, as in HP C, except arrays and functions which are passed as pointers.

HP FORTRAN 77 passes all arguments by reference. This means that all actual parameters in an HP C++ call to a FORTRAN routine must be pointers, or variables prefixed with the unary address operator `&`.

HP Pascal passes arguments by value, unless specified as `var` parameters. There are two ways to pass variables to Pascal `var` parameters. One way is to use the address operator `&`. The other way is to declare the variable as a pointer to the appropriate type, assign the address to the pointer, and pass the pointer.

So, the simplest way to reconcile these 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 example illustrates a reference variable.

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

Refer to "References" in *The C++ Programming Language* for details about using reference variables.

Using extern "C" Linkage

If you want to mix C++ modules with HP FORTRAN 77 or HP Pascal 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 or Pascal routines.

Strings

HP C++ strings are not the same as HP FORTRAN 77 strings. In FORTRAN 77 the strings are not null terminated. Also, strings are passed as string descriptors in FORTRAN 77. This means that the address of the character item is passed and a length by value follows.

Note On the HP 9000 Series 700/800, the length follows immediately after the character pointer in the parameter list.

HP Pascal strings and HP C++ strings are not compatible. See your HP Pascal manual for details.

Arrays

HP C++ stores arrays in row-major order, whereas HP FORTRAN 77 stores arrays in column-major order. The lower bound for HP C++ is 0. The default lower bound for HP FORTRAN 77 is 1. For HP Pascal, the lower bound may be any user-defined scalar value.

Definition of TRUE and FALSE

On the HP 9000 Series 700/800, HP C++, HP FORTRAN 77, and HP Pascal do not share a common definition of TRUE or FALSE. HP C++ does not have a FORTRAN LOGICAL type. Instead C++ uses integers. In HP C++, any nonzero value is used to represent TRUE and 0 is used to represent FALSE.

HP C++ does not have a Pascal boolean type. On the HP 9000 Series 700/800, HP Pascal allocates 1 byte for boolean variables and only accesses the rightmost bit to determine its value, 1 to represent TRUE and 0 for FALSE.

Files

HP FORTRAN I/O routines require a logical unit number to access a file, whereas HP C++ accesses files using HP-UX I/O subroutines and intrinsics and requires a stream pointer.

A FORTRAN logical unit cannot be passed to a C++ routine to perform I/O on the associated file. Nor can a C++ file pointer be used by a FORTRAN

routine. However, a file created by a program written in either language can be used by a program of the other language if the file is declared opened within the latter program. HP-UX I/O (stream I/O) can also be used from FORTRAN instead of FORTRAN I/O.

Refer to your system FORTRAN manual on inter-language calls for details.

A C++ file pointer cannot be passed to a Pascal routine for performing input/output. A Pascal file variable cannot be used by a C++ program. However, a file created by a program written in either language can be used by a program of the other language if the file is declared opened within the latter program.

If I/O from Pascal is required, it is recommended that you use HP-UX input/output routines and intrinsics. This allows C++ and Pascal to use the same I/O mechanism.

See the HP Pascal manual for your system for more details.

Linking HP FORTRAN 77 and HP Pascal Routines on HP-UX

When calling HP FORTRAN 77 or HP Pascal routines on the HP 9000 Series 700/800, you must include the appropriate run-time libraries by adding the following argument to the `CC` command:

```
-lcl -lisamstub
```

6

HP Specific Features of lex and yacc

Following is a list of HP specific features of `lex` and `yacc`. For more information on these tools, see the `lex` and `yacc` man pages or the *HP-UX Reference*. Another general source of information is *lex and yacc* by John R. Levine, Tony Mason, and Doug Brown.

- `LC_CTYPE` and `LC_MESSAGES` environment variable support in `lex` - Determines the size of the characters and language in which messages are displayed while you use `lex`.
- `-m` command line option for `lex` - Specifies that multibyte characters may be used anywhere single byte characters are allowed. You can intermix both 8-bit and 16-bit multibyte characters in regular expressions if you enable the `-m` command line option.
- `-w` command line option for `lex` - Includes all features in `-m` and returns data in the form of the `wchar_t` data type.
- `%l <locale>` directive for `lex` - Specifies the locale at the beginning of the definitions section. Any valid locale recognized by the `setlocale` function can be used. This directive is similar to using the `LC_CTYPE` environment variable. To receive `wchar_t` support with `%l`, use the `-w` command line option.
- `LC_CTYPE` environment variable support in `yacc` - Determines the native language set used by `yacc` and enables multibyte character sets. Multibyte characters can appear in token names, on terminal symbols, strings, comments, or anywhere ASCII characters can appear, except as separators or special characters.

Notes on Using `lex` and `yacc`

When using `lex` and `yacc`, please note the following:

- Programs generated by `yacc` or `lex` can have many unreachable `break` statements, causing multiple C++ warnings.
- If you want to call the yacc generated routines, yyerror, yylex and yyparse, your program must include the yacc.h header file.

```
#include <yacc.h>
```

Index

D

E

F

G

H

I

K

L

R

S

T