HEWLETT PACKARD

HP 9000
Series 300, 400,
700 and 800
Computers

HP C++
Programmer's Guide

HP 9000 Computers

HP C++ Programmer's Guide

Series 300/400 and 700/800 Systems

HP Part No. 92501-90005
Printed in U.S.A. August 1992

Third Edition
E0892

Notice

UNIX is a registered trademark of UNIX System Laboratories Inc. in the USA and other countries.

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)

Documentation Map

The following lists additional documentation you may find useful. You can order any of these HP books from HP DIRECT by calling 1-800-538-8787 and providing the HP part number.

Documentation for the Relatively New User

Document	Part Number	Description
A Beginner's Guide to HP-UX	HP B1862-90000	Introductory information about using the HP-UX operating system.
Shells: User's Guide	HP B2355-90046	Information about using command shells on HP-UX.
The Ultimate Guide to the vi and ex Text Editor	HP 97005-90015	Complete information about using the vi and ex text editors on HP-UX.
HP-UX Reference Manual	HP B2355-90033	A complete description of all HP-UX commands.
Programming on HP-UX[1]	HP B2355-90026	A description of the fundamentals of software development on HP-UX, including managing both archived and shared libraries, linking and running programs, managing versions of source code, using various system libraries for input and output, character string, date, and time manipulation, and mathematical functions, and maintaining programs with the **make** command.
C++ Primer, second edition, by Stanley Lippman	ISBN 0-201-54848-8 Not available from HP	A complete tutorial introduction to C++, for those with little or no C or C++ experience. This book is available at technical bookstores.

1 This book comes with the HP C product documentation.

Documentation for the Relatively Advanced User

Document	Part Number	Description
The C++ Programming Language, second edition, by Bjarne Stroustrup[1]	HP B2402-90001 ISBN 0-201-53992-6	A tutorial on C++ including a complete language reference manual and information about object-oriented design and software development. This book is also available at technical bookstores.
The Annotated C++ Reference Manual, by Margaret Ellis and Bjarne Stroustrup	ISBN 0-201-51459-1 Not available from HP	A complete C++ language reference manual plus annotations and commentary that describe in detail why features are defined as they are. This book is available at technical bookstores.
C++ Language System Product Reference Manual	Available from USL	This book is not included with HP C++ documentation because *The C++ Programming Language* described above contains the complete *Reference Manual.*
C++ Language System Selected Readings[1]	Available from USL	Topical discussions about C++, including an overview of C++, multiple inheritance, parameterized types, and type-safe linkage. See below for ordering information.
C++ Language System Library Manual[1]	Available from USL	Descriptions of the `complex`, `task`, and `iostream` class libraries provided with release 3.0 of HP C++. See below for ordering information.
C++ Language System Release Notes[1]	Available from USL	Information on compatibility with previous releases, future compatibility issues, and known problems. See below for ordering information.

1 This book comes with the HP C++ product documentation.

Documentation for the Relatively Advanced User (continued)

Document	Part Number	Description
HP-UX Symbolic Debugger User's Guide[1]	HP B1864-90005 (for HP-UX 8.0) HP B2355-90044 (for HP-UX 9.0)	Information on debugging C++ programs on the HP 9000.
HP-UX Symbolic Debugger Technical Addendum[2]	HP B2355-90017 (for HP-UX 8.0 only)	An addendum to the *HP-UX Symbolic Debugger User's Guide*, this contains information not in the *HP-UX Symbolic Debugger User's Guide*, about debugging C++ programs.
HP C++ Programmer's Guide (this book)[2]	HP 92501-90005	Information about C++ programming on the HP 9000, including using the `CC` command, using the preprocessor, optimizing your programs, calling programs written in other languages, and using the `lex` and `yacc` programs.
C++ Quick Reference Card[2]	HP B1637-90001	A quick, convenient summary of C++ syntax and concepts, including templates and exception handling.
Codelibs Library Reference - Version 2.100[2]	HP B2617-90000	Complete information on the HP Codelibs class library.
USL C++ Standard Components Manual[2]	USL C308	Complete information on the USL Standard Components class library.

1 This book comes with both the HP C++ and the HP C product documentation.

2 This book comes with the HP C++ product documentation.

Documentation for the C User

Document	Part Number	Description
HP C/HP-UX Reference Manual[1]	HP 92453-90024	Detailed information on the C language for the HP 9000 Series 700/800 systems.
C, A Reference Manual, third edition, by Samuel Harbison and Guy Steele[1]	HP B1700-90002; ISBN 0-13-110933-2	Detailed information on the C language for the HP 9000 Series 300/400 systems. This book is also available at technical bookstores.
HP C Programmer's Guide[1]	HP 92434-90002 for Series 700/800 and HP B1864-90008 for Series 300/400 systems	For advanced information on using HP C on the HP 9000.

1 This book comes with the HP C product documentation.

The following documents are also referenced in this manual.

- *HP-UX Assembler and Tools* (B1864-90014 for HP-UX 9.0)
- *HP-UX Floating Point Guide* (B2355-90624 for HP-UX 9.0 only)

To order additional copies of any of the USL books, contact USL in the USA at 1-800-828-UNIX (1-800-828-8649) or contact your local AT&T or USL sales office. Some sales offices are listed in appendix A of the *USL C++ Standard Components Getting Started* manual. Provide the USL product code or title of the book you want.

Preface

The *HP C++ Programmer's Guide* was written to assist C and C++ programmers execute and debug C++ programs on HP 9000 Series 300/400 and 700/800 computers. 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*. Use the "Documentation Map" to decide which sources to consult.

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 — Errors and Diagnostic Messages provides general information on error messages generated by the HP C++ compiling system.

Chapter 7 — lex: A Lexical Analyzer and Generator describes the `lex` command, a program generator designed for lexical processing of character input streams.

Chapter 8 — yacc: Yet Another Compiler-Compiler describes the `yacc` command, a general tool for describing the input to a computer program.

Conventions

NOTATION	DESCRIPTION
(see margin)	Change bars in the margin show where substantial changes have been made since the last edition.
`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*] [,...]

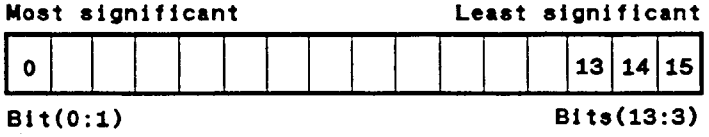
... ⋮ Within examples, horizontal or vertical ellipses indicate where portions of the example are omitted.

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 (UNIX System Laboratories, Inc.) 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 (UNIX System Laboratories, Inc.) 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).

For more information refer to "Differences between HP C++ Version 3.0 and Version 2.1" in this chapter.

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`.

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. Also refer to "Differences between HP

C++ Version 3.0 and Version 2.1" and the sections following it for information about the differences between the current and previous releases of the USL C++ translator. 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 `+OV` option makes all global variables `volatile`, and performs level 2 optimization. See Chapter 3 for more information about the `+OV` option.

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 it 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.

```
//*************************************************************
//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.

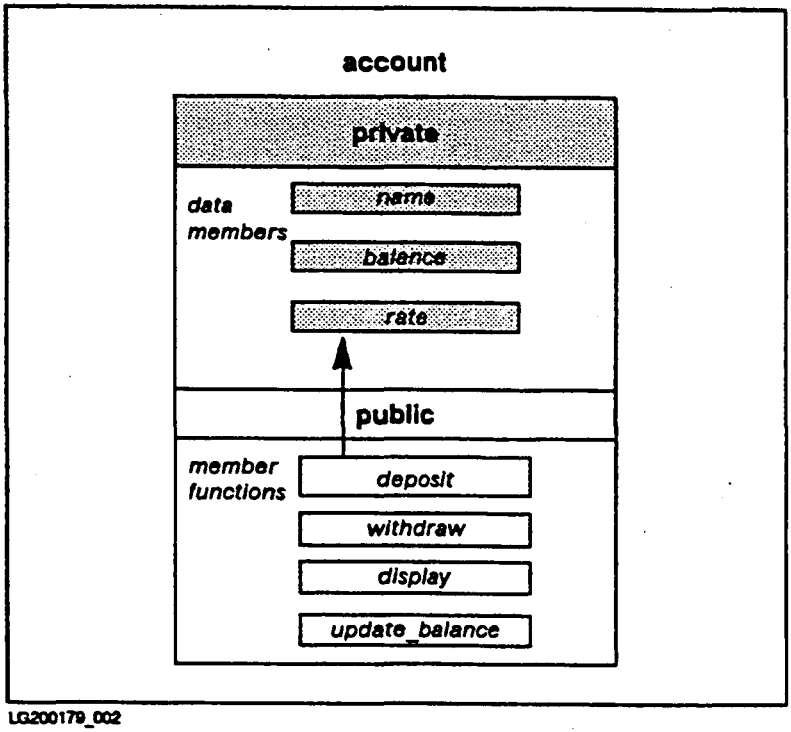


Figure 1-1.
Encapsulation in a C++ Class: The account class Example

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.

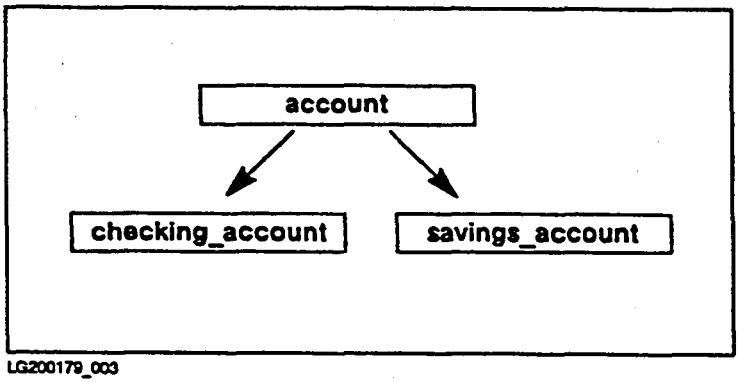


Figure 1-2. Concept of Single Inheritance: The account Example

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.

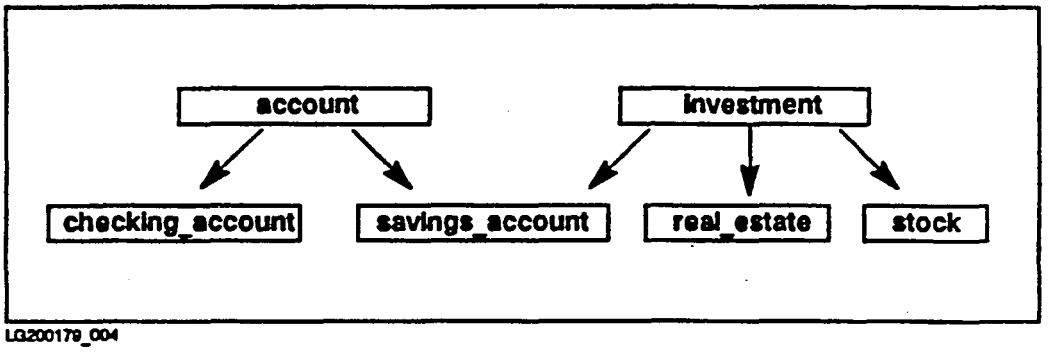


Figure 1-3. Concept of Multiple Inheritance: The savings_account Example

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*.

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 adds 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
```

Differences between HP C++ Version 3.0 and Version 2.1

Version 3.0 of HP C++ adds significant new functionality and fixes a number of problems.

New Features of HP C++ 3.0

Version 3.0 of HP C++ adds the following new features:

- templates
- exception handling
- complete implementation of nested types
- capability to use a constructor with all default arguments as a default, or void, constructor
- capability to overload both the prefix and postfix increment (++) and decrement (--) operators
- `protected` derivation
- extension of the dominance rule to include dominance of data and enumerators as well as functions

For more information about exception handling, see the *The C++ Programming Language* and the *HP C++ Release Notes*. For more information about the other features see "New Features" in chapter 4 of the *C++ Language System Release Notes*. For the latest information about HP C++, see the online *HP C++ Release Notes* file `/usr/contrib/80RelNotes/HPCXX.7` for HP-UX 8.0 and `/usr/contrib/90RelNotes/HPCXX` for HP-UX 9.0.

HP C++ Version 3.0 Detects More Errors

Version 3.0 of HP C++ more strictly conforms to the definition of C++ as described in both the *The C++ Programming Language* and *The Annotated C++ Reference Manual*. This means that certain illegal constructs that were silently accepted by version 2.1 will now cause errors. Some of these cases are described in chapter 4 under "Language-Related Fixes" of the *C++ Language System Release Notes*.

In addition, illegal code that caused warnings in version 2.1 now cause compile-time errors in version 3.0. For example, if you used nested types and had compiler warnings, you will get errors in version 3.0.

For complete information about differences between version 3.0 and previous versions, and for information on what is likely to change in future versions of C++, refer to chapter 4, "Compatibility", in the *C++ Language System Release Notes*.

Link Compatibility

C++ programs compiled with version 3.0 will link with programs compiled with version 2.1, unless you are using exception handling. If *any* of your files uses exception handling, you must recompile *all* your program files, including your libraries, with version 3.0 and the `+eh` option. If you attempt to link some object files compiled with `+eh` and some object files not compiled with `+eh`, `c++patch` will issue an error and will not link the program.

Future Link Compatibility

The next major revision of HP C++ will most likely *not* be link-compatible with version 3.0. That is, you will probably have to recompile all your C++ programs to use the next major revision of HP C++.

Compatibility between Translator Mode and Compiler Mode

Compiler mode and translator mode are completely compatible with respect to C++ language functionality, object files, symbolic debug information, header files, libraries, and program development tools. To migrate your C++ source code from translator mode to compiler mode, simply recompile and relink. Since both translator mode and compiler mode object files will link with each other, you can migrate all or part of your application. If you are using exception handling, see "Link Compatibility" earlier in this chapter.

Compiler Mode is the Default

Since compiler mode is the default, use the `+T` option of the `CC` command to request translator mode.

The Stream Library

The stream library, which is not part of the C++ language definition but which provides facilities for I/O, was rewritten by AT&T USL for its 2.0 release of C++. HP C++ uses the stream library shipped with the USL release 3.0 of the translator. Refer to the "Compatibility" chapter in the *C++ Language System Release Notes* for a summary of the changes in the stream library, and to the "Iostream Examples" chapter in the *C++ Language System Library Manual*, for examples using the library. C++ programs using the stream library should use the following preprocessor directive:

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

The 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."

HP Symbolic Debugger

The HP Symbolic Debugger fully supports C++ templates and exception handling. For more information see the *HP-UX Symbolic Debugger User's Guide* and the *HP-UX Symbolic Debugger Technical Addendum.*

Differences between HP C++ Version 2.1 and Version 2.0

Version 2.0 of HP C++ translated C++ source code to C source code, then compiled the C source code using the C compiler. Version 2.1 added compiler mode to HP C++. Compiler mode, the default, compiles your C++ source code directly to object code, providing significantly faster compiles.

For more information about differences between version 2.1 and previous versions, refer to chapter 4, "Compatibility", in the *C++ Language System Release Notes*.

Differences and Incompatibilities between Version 1.2 and HP C++ Version 2.0

Version 2.0 of the HP C++ translator is not fully source compatible or link compatible with releases of the USL translator before version 2.0. As a result, users of earlier versions of C++ probably need to make changes in their source code and also recompile. These differences and incompatibilities are discussed briefly in the rest of this chapter. If you are not migrating from an earlier version, the information in this section is probably not relevant to you.

Language Incompatibilities and Differences

HP C++ language features that are not part of previous releases of the USL translator include the following:

- multiple inheritance
- abstract class specification
- recursive memberwise initialization and assignment
- implicit overloading and type safe linkage to other languages
- overloading of operators `new` and `delete`
- improved mechanism for overloaded function matching (withdrawal of the keyword `overload`)
- additional reserved keywords: `template`, `catch`, and `volatile`

- support for `static` and constant member functions
- support for declarations of static arrays of class objects that do not have default constructors
- support for the operator `new` to create array types

For greater detail about these language features, see *The C++ Programming Language*.

In addition to new language features, the HP C++ compiler now issues error messages rather than warnings for certain illegal language constructs. It also issues additional warnings for anachronisms. Refer to chapter 4, "Compatibility with Previous Releases," of *C++ Language System Release Notes* for more details about diagnostic messages, as well as a complete list of language extensions and changes.

Linkage with Other Languages

Previous releases of the USL C++ translator had problems dealing with the name ambiguity that can arise when you link C++ source files with C library files. Releases 2.0 of HP C++ addressed these problems by supporting implicit (rather than explicit) overloading of function names and by supporting C++ (rather than C) linkage as the default. This means you must explicitly declare a function to have C, rather than C++, linkage. You do this using the `extern "C"` declaration. Refer to Chapter 5, "Inter-Language Communication," of this document for more information and program examples using the `extern "C"` specification in HP C++ source files.

Note All HP C++ header files supplied with versions 2.1 and later contain the necessary `extern "C"` declarations. If you use any user-created header files from previous releases, be sure to convert any declarations of C function names to the new `extern "C"` syntax.

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
- 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.

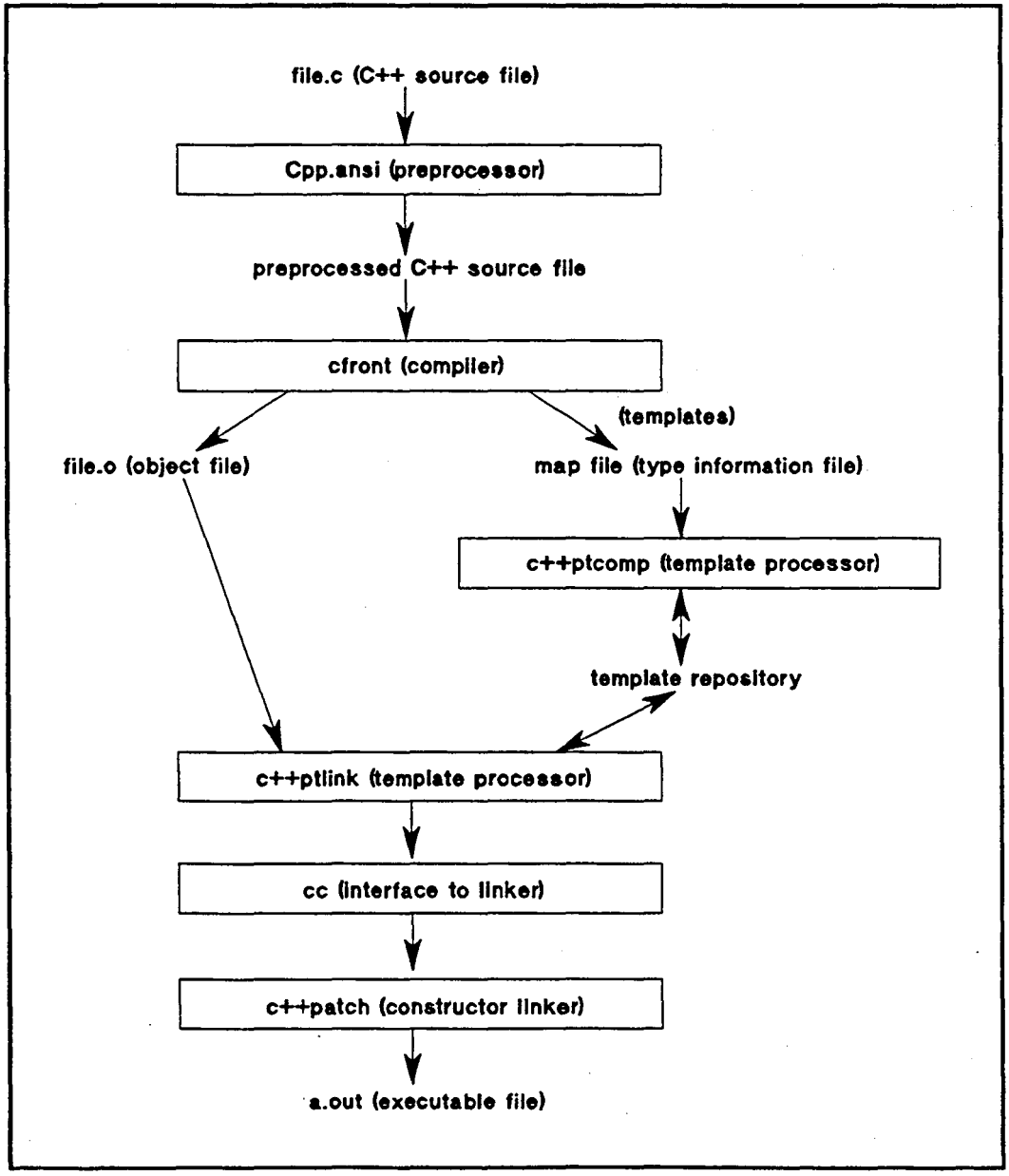


Figure 3-1. Phases of the HP C++ Compiling System in Compiler Mode

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. (On Series 300/400 systems, when you request level 2 optimization, the compilation phase runs `cfront2` rather than `cfront`.)

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` 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.

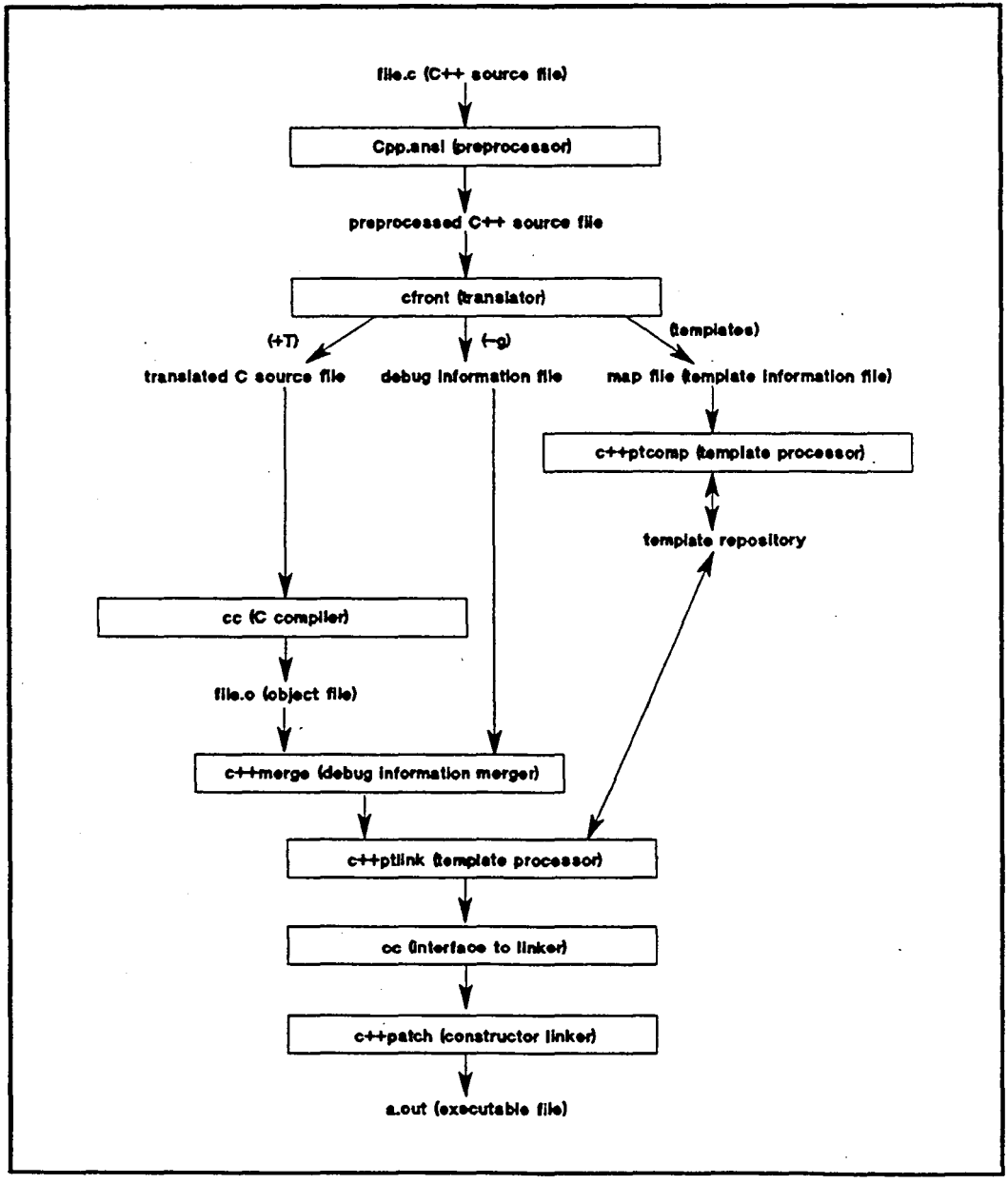


Figure 3-2. Phases of the HP C++ Compiling System in Translator Mode

Phases of the Compiling System

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` 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.

3

Setting Your Path to the CC Command

The CC command is normally installed in the directory **/usr/bin**. 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. See Table 3-2 for additional options supported by HP C++ on Series 300/400 systems and Table 3-3 for additional options supported on Series 700/800 systems.

3

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 *Programming on HP-UX*.
`-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
`-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. This option is incompatible with optimization. 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/libx.sl` or `/lib/libx.a` then `/usr/lib/libx.sl` or `/usr/lib/libx.a` 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*.
`-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 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.
`-Z`	Allows dereferencing of null pointers at run time. The value returned from a dereferenced null pointer is zero.

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.
`+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.
`+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.
`+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.
`+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
`+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`*opt*	Invokes the level of optimization selected by *opt*. *opt* can have any of the following values: `1` Performs level 1 optimization only. `2` Performs level 2 optimization. `3` Performs level 3 optimization. (Series 700 and 800 only.) `V` Performs level 2 optimization. In addition, all global variables are treated as if they were declared with the keyword `volatile`, meaning that references to the object cannot be optimized away. For more information on optimization, see Chapter 4 "Optimizing HP C++ Programs."
`+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.
`+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.

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

Option	Effect of Specifying the Option
+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.
+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 *Programming on HP-UX*.
+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 in versions 2.0, 2.1, and 3.0 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.

Series 300/400 Compiler Options

Table 3-2 presents additional options supported by CC on Series 300/400 systems.

Table 3-2.
Additional CC Command Options on the Series 300/400

Option	Effect of Specifying the Option
`+bfpa`	Causes the compiler to generate code that uses the HP 98248A floating point accelerator card, if the card has been installed before the program is run. If the card has not been installed, floating-point operations are done on the MC68881 math coprocessor.
`+ffpa`	Causes the compiler to generate code for the HP 98248A floating-point accelerator card. This code does not run unless the card is installed.
`+M`	Causes the compiler not to generate inline code for the MC68881 math coprocessor. The compiler generates calls to math library routines instead.
`-t`*x*, *name*	Substitutes or inserts subprocess *x* using *name*. (See Table 3-1 for complete information on using this option.) You can use the following subprocess identifiers in addition to those presented in Table 3-1. **Value** — **Description** `0` — The optimizing version of the C++ compiler. (standard suffix is `cfront2`) `0` (zero) — First pass of the compiler with level 2 optimization. This is not the same as subprocess `c` (standard suffix is `cpass1`). Valid in translator mode only. `1` — Second pass of the compiler with level 2 optimization (standard suffix is `cpass2`). `g` — Level 2 global optimizer (standard suffix is `c.c1`). `2` — Peephole optimizer (standard suffix is `c.c2`).

Note On the HP 9000 Series 300/400 the default is to allow null pointer dereferencing, so using `-Z` has no effect.

Series 700/800 Compiler Options

Table 3-3 presents additional options supported by CC on Series 700/800 systems. The *+optionname [optionarg]* notation can be used as well.

Table 3-3.
Additional CC Command Options on the Series 700/800

Option	Effect of Specifying the Option
`-z`	Disallows dereferencing of null pointers at run time. Fatal errors result if null pointers are dereferenced.
`+DA`*model*	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` or `-lM`. (HP-UX 9.0 only) (See the *HP-UX Floating-Point Guide* for more information about using math libraries.) *model* 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` or `1.1`. For example, specifying either `+DA1.0` or `+DA845` generates code for the PA-RISC 1.0 architecture because the model 845 is based on PA-RISC 1.0. Similarly, 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. Use the command `uname -m` to determine the model number of your system. Note: Object code generated for PA-RISC 1.1 will *not* execute on PA-RISC 1.0 systems. So for portability use `+DA1.0` and for best performance use `+DA` with the model number or architecture where you plan to execute the program. If you do not specify this option, the default object code generated is PA-RISC 1.1 when compiling on any Series 700 system and PA-RISC 1.0 when compiling on any Series 800 system. For more information on this option, see Chapter 4.

Table 3-3.
Additional CC Command Options on the Series 700/800
(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` or `1.1`. 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` or `+DS1.1` performs scheduling for a representative PA-RISC 1.0 or 1.1 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. 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. For more information on this option, see Chapter 4.

Table 3-3.
Additional CC Command Options on the Series 700/800
(continued)

Option	Effect of Specifying the Option
+Obb*num*	Performs level 2 optimization but only on functions with *num* or fewer basic blocks. (A basic block is a sequence of code with a single entry point, single exit point, and no internal branches.) Any function with more than *num* basic blocks is optimized at level 1 instead and a warning message is generated giving the name of the function and the number of basic blocks it contains. If you do not use this option, the default maximum number of basic blocks per function is 500. In other words, all functions with 500 or fewer basic blocks are optimized at level 2, and all functions with over 500 basic blocks are optimized at level 1. For more information on optimization, see Chapter 4, "Optimizing HP C++ Programs."
+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.

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 CXXOPTS environment variable 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.

For example, the command below causes the option -v to be passed to the CC command each time you execute the CC command. The following commands set the CXXOPTS variable:

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

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

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

Caution Using the CCOPTS environment variable in translator mode can cause unexpected side effects to the HP C++ compilation process. Because in translator mode HP C++ uses the C compiler (cc) for code generation, options passed to cc by the CC command could conflict with options specified by the CCOPTS variable.

Environment Variables: TMPDIR, CCLIBDIR, CCROOTDIR

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 as the default directory for temporary files. The syntax for TMPDIR in csh notation is

3

```
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 alternate the 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

This section lists pragmas that affect how optimization is done. For more information on optimization, see Chapter 4, "Optimizing HP C++ Programs."

Pragma OPTIMIZE.

```
#pragma OPTIMIZE { ON
                 { OFF
```

The `OPTIMIZE` pragma turns on or off level 2 and 3 optimization. (Level 3 optimization is supported on Series 700 and 800 systems only.) It does not turn off level 1 optimization. It is useful for turning off level 2 and 3 optimization in sections of a source program that may cause the optimizer difficulties. You must have specified either the `-O` option or the `+O` option on the CC command. Otherwise this pragma is ignored. This pragma may not be used within a function.

Pragma OPT_LEVEL.

```
#pragma OPT_LEVEL { 1
                  { 2
                  { 3
```

The `OPT_LEVEL` pragma sets the optimization level to 1, 2, or 3. You must have specified either the `-O` option or the `+O` option on the CC command. Otherwise this pragma is ignored. This pragma may not be used within a function. See Table 3-1 for more information on the `-O` and `+O` options.

Pragmas for Shared Libraries

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

Pragma HP_SHLIB_VERSION.

3

```
#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 *Programming on HP-UX*.

Pragmas on Series 700/800 Systems

The following pragmas may be used only on the HP 9000 Series 700/800. Any other pragmas not recognized by the HP C++ compiler generate a warning to `stderr`.

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`; 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 (`/lib/libc.sl` and `/lib/libc.a`), 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*.

3

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`. 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*.

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`. 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` 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 3-4 lists the files containing the HP C++ run-time libraries. Different libraries are used depending on whether or not you use exception handling.

Table 3-4. 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`

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`. 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`. 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 *Programming on HP-UX*. 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:

$$\texttt{-Wl,-a,}\begin{Bmatrix}\texttt{archive}\\ \texttt{shared}\\ \texttt{default}\end{Bmatrix}$$

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.

3

```
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 *Programming on HP-UX*.

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 *Programming on HP-UX* 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 *Programming on HP-UX* 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 *Programming on HP-UX* for information about `shl_load()`, including syntax. `cxxshl_unload()` can be used on non-C++ shared libraries and can be called from other languages.

3

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`. 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 *Programming on HP-UX*.

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 C++ Libraries, Object Files, and Executable Files

3

This section describes what you need to do if you write applications in HP C++ and distribute any of the following C++ files to your customers:

- archived libraries containing C++ code
- shared libraries containing C++ code
- object files produced by HP C++
- executable files produced by HP C++
- applications that use shared libraries provided with HP C++
- any combination of the above

This section explains what you need to do to ensure that your customers can use your code correctly.

Applications That Use HP C++ Shared Libraries

If your application uses any of the shared libraries that come with HP C++, your customer must have those libraries installed on their system to run the application. If your customer already has the necessary HP C++ shared libraries installed, the application will work. Otherwise, you need to distribute them to your customers. See "HP C++ Files You May Distribute" later in this chapter for a list of the files you may distribute.

Linking with C++ Libraries

This section discusses what you and your customers need to do if your product is a C++ library.

The C++ language requires that nonlocal static objects be initialized before any function or object is used. HP C++ initializes nonlocal static objects in all object files, including shared libraries, before the first statement in `main()` executes. If you distribute C++ libraries that your customers will use, they must do the following to ensure that nonlocal static objects are correctly initialized and destroyed:

- Your customer's `main` program must first call `_main()` before doing anything else. `_main()` calls all nonlocal static constructors.

- Your customer's program must be linked with the `CC` command. You may distribute the `CC` driver program along with the libraries it needs. See the section "HP C++ Files You May Distribute" for these libraries.
- If the program explicitly loads or unloads a C++ shared library, you must use `cxxshl_load` and `cxxshl_unload` instead of the standard `shl_load` and `shl_unload` routines. `cxxshl_load` and `cxxshl_unload` ensure that constructors and destructors of nonlocal static objects in the shared library are called.

In addition, your customers should review Chapter 5, "Inter-Language Communication," for information on linking HP C++ modules with HP C, HP Pascal, and HP FORTRAN 77.

Installing Your Application

Your application's installation procedure should install the appropriate HP C++ components in the standard places on your customer's systems. This will ensure that the `CC` command can find them.

If your customer already has HP C++ installed and their version is newer than yours, you should not overwrite any of the existing HP C++ components. In addition, you should not install your product on a system that has a newer version of HP C++ if that newer version is incompatible with your version.

You should also warn your customers not to install a version of HP C++ after installing your product if their version of HP C++ is incompatible with your version.

HP C++ releases are usually upward compatible, but HP cannot guarantee that this will be true for all releases. If you have questions about the compatibility of HP C++ releases, you should contact your HP support representative.

HP C++ Files You May Distribute

For this release only, HP grants you permission to package and redistribute the following subset of HP C++ components to your customers:

3

- Libraries:

`/usr/lib/libcxx.a`	Always distribute this.
`/usr/lib/libC.a`	Distribute this if your code was directly compiled or was translated in K&R C mode and you want the archived library.
`/usr/lib/CC/eh/libC.a`	Distribute this if your code was directly compiled or was translated in K&R C mode and you use exception handling.
`/usr/lib/libC.sl`	Distribute this if your code was directly compiled or was translated in K&R C mode and you want the shared library.
`/usr/lib/libC.ansi.a`	Distribute this if your code was compiled or translated in ANSI C mode and you want the archived library.
`/usr/lib/CC/eh/libC.ansi.a`	Distribute this if your code was compiled or translated in ANSI C mode and you use exception handling
`/usr/lib/libC.ansi.sl`	Distribute this if your code was compiled or translated in ANSI C mode and you want the shared library.
`/usr/lib/libcomplex.a`	Distribute this if your code uses it.
`/usr/lib/CC/eh/libcomplex.a`	Distribute this if your code uses it and you use exception handling.
`/usr/lib/libtask.a`	Distribute this if your code uses it.
`/usr/lib/libOstream.a`	Distribute this if your code uses it.
`/usr/lib/libdemangle.a`	Distribute this if your code uses it.
`/usr/lib/libcodelibs.a`	Distribute this if your code uses it.
`/usr/lib/libcodelibs.sl`	Distribute this if your code uses it.
`/usr/lib/CC/eh/libcodelibs.a`	Distribute this if your code uses it and you use exception handling.
`/usr/lib/lib++.a`	Distribute this if your code uses it.
`/usr/lib/CC/eh/lib++.a`	Distribute this if your code uses it and you use exception handling.
`/usr/lib/libGA.a`	Distribute this if your code uses it.

`/usr/lib/CC/eh/libGA.a`	Distribute this if your code uses it and you use exception handling.
`/usr/lib/libGraph.a`	Distribute this if your code uses it.
`/usr/lib/CC/eh/libGraph.a`	Distribute this if your code uses it and you use exception handling.
`/usr/lib/libfs.a`	Distribute this if your code uses it.
`/usr/lib/CC/eh/libfs.a`	Distribute this if your code uses it and you use exception handling.
`/usr/lib/libg2++.a`	Distribute this if your code uses it.
`/usr/lib/CC/eh/libg2++.a`	Distribute this if your code uses it and you use exception handling.
`/usr/lib/incl2`	Distribute this if your code uses it.
`/usr/lib/CC/eh/incl2`	Distribute this if your code uses it and you use exception handling.
`/usr/lib/hier2`	Distribute this if your code uses it.
`/usr/lib/CC/eh/hier2`	Distribute this if your code uses it and you use exception handling.
`/usr/lib/publik2`	Distribute this if your code uses it.
`/usr/lib/CC/eh/publik2`	Distribute this if your code uses it and you use exception handling.

- Executable files:

`/usr/bin/c++filt`	Always distribute this.
`/usr/lib/c++patch`	Always distribute this.
`/usr/bin/CC`	Always distribute this.

- Error message catalogs:

`/usr/lib/nls/C/CC.cat`	Always distribute this.

- Manual pages:

`/usr/man/man1.Z/CC.1`	Always distribute this.

Terms for Distribution of HP C++ Files

Permission to distribute the above mentioned HP C++ components is based on the following terms and conditions:

1. These HP C++ components cannot be redistributed as part of a C++ compiler, translator, linker or interpreter product.

2. All copyright notices in the code must be retained.
3. The HP C++ executable components can only be redistributed by HP C++ customers.

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`.) 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 Library Example

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`.

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

This chapter describes how HP C++ programs can be optimized for improved efficiency. HP C++ provides options to the CC command and pragmas to control optimization.

Levels of Optimization

The HP C++ compiler provides four levels of optimization (levels 0, 1, 2, and 3) as well as other specific kinds of optimizations. Table 4-1 summarizes each optimization level, gives the advantages and disadvantages of using the level, and recommends when to use the level.

Table 4-1. C++ Optimization Levels

Level	Optimizations Performed	Advantages	Disadvantages
0	Constant folding, simple register assignment.	Compiles fastest. Works with debugger options `-g` and `-g1`.	Does minimal optimization.
1	Level 0 optimizations, plus peephole (statement-by-statement) optimizations and instruction scheduling.	Produces faster programs than level 0. Compiles faster than level 2.	Compiles slower than level 0. Does not work with debugging options `-g` and `-g1`.
2	Level 1 optimizations, plus global (whole procedure) optimizations.	Produces faster run-time code than level 1.	Compiles slower than level 1. Does not work with debugging options `-g` and `-g1`.
3	Level 2 optimizations, plus procedure integration.	Produces the fastest run-time code.	Compiles slower than level 2. May increase program size and does not work with debugging options `-g` and `-g1`.

Requesting Optimization

By default, the HP C++ compiler performs level 0 optimization. HP C++ provides two ways to control the level of optimization performed on your program:

- the `-O` and `+O` options to the `CC` command
- pragmas included in your source programs

In addition, you can use the `+DA` and `+DS` options on Series 700/800 systems to optimize the instruction set and scheduling for a particular implementation of the PA-RISC (Precision Architecture-Reduced Instruction Set Computer) architecture. See "Compiling for Different Versions of the PA-RISC Architecture" later in this chapter for more information.

Options to CC That Request Optimization

Table 4-2 briefly lists the options to the CC command that request optimization. For a complete description of these options, see Chapter 3.

Table 4-2. Optimization Options

Option	Systems Where Option Can Be Used	What the Option Does
-O	300/400, 700/800	Requests level 2 optimization for the file being compiled.
+O1	300/400, 700/800	Requests level 1 optimization for the file being compiled.
+O2 or -O	300/400, 700/800	Requests level 2 optimization for the file being compiled.
+O3	700/800	Requests level 3 optimization for the file being compiled.
+OV	300/400, 700/800	Requests level 2 optimization for the file being compiled and ensures that no global memory references are optimized away.
+Obb*num*	700/800	Performs level 2 optimization but only on functions with *num* or fewer basic blocks.

You can specify any of these options on the command line or in the CXXOPTS environment variable.

Compiling for Different Versions of the PA-RISC Architecture

Different HP 9000 systems use different versions of the PA-RISC architecture. Some models use PA-RISC 1.0 while other models use PA-RISC 1.1. The instruction set on PA-RISC 1.1 is a superset of the instruction set on PA-RISC 1.0. As a result, code generated for PA-RISC 1.0 systems will run on PA-RISC 1.1 systems, though possibly less efficiently than if it were specifically generated for PA-RISC 1.1. However, code generated for PA-RISC 1.1 systems will *not* run on PA-RISC 1.0.

4 By default, compiling on any Series 800 system generates PA-RISC 1.0 code and compiling on any Series 700 system generates PA-RISC 1.1 code. Use the `+DA` option to change this default behavior. See Table 3-3 for more information about the `+DA` option.

In addition, the instruction scheduling is different on some implementations of these architectures. You can improve performance on a particular model of the HP 9000 by requesting that the compiler use instruction scheduling tuned to that particular model. However, in contrast with the different instruction sets discussed above, using scheduling for one model does not prevent your program from executing on another model.

By default, the compiler uses scheduling tuned for the system where you are compiling. Use the `+DS` option to change this default behavior. See Table 3-3 for more information about the `+DS` option.

Using +DA to Generate Code for a Specific Version of PA-RISC

Use the `+DA` option to specify which PA-RISC instruction set the compiler should use when generating code. Specifying `+DA1.0` ensures your code will run on all HP 9000 models, although the performance of your program may not be as good as it could be on PA-RISC 1.1 systems. Specifying `+DA1.1` may give better performance on PA-RISC 1.1 systems, but the executable file generated with this option will not run on PA-RISC 1.0 systems.

Using +DS to Specify Instruction Scheduling

Use the `+DS` option to specify instruction scheduling tuned to a particular implementation of PA-RISC. For example, to specify instruction scheduling for the model 867, use `+DS867`.

Guidelines for Using +DA and +DS

When you use the +DA and +DS options depends on your particular circumstances. Here are some possibilities.

- If you plan to run your program on the same system where you are compiling, you don't need to use either the +DA or +DS option. The compiler generates code tuned for your system.
- If you plan to run your program on one particular model of the HP 9000 and that model is different from the one where you compile your program, use the following combination:
  - +DA*model* with the model number of the target system, and
  - +DS*model* with the model number of the target system.

  For example, if you are compiling on a 720 and your program will run on an 855, you should use +DA855 +DS855. This will give you the best performance on the 855.
- If you plan to run your program on many models of the HP 9000, you could use the following combination:
  - +DA1.0 to ensure portability, and
  - +DS*model* with the model number of the fastest system you will be running your application on.

  For example, using +DA1.0 +DS897 ensures your program can run on all Series 700s and 800s, and uses scheduling for the model 897. You might want to use scheduling for a high-performance system (such as the 897), assuming your customers with high-performance systems want the fastest performance from your application.

Compiling in Networked Environments

When compiles are performed using diskless workstations or NFS-mounted file systems, it is important to note that the default code generation and scheduling are based on the local host processor. The system model numbers of the hosts where the source or object files reside do not affect the default code generation and scheduling.

Pragmas That Control Optimization

Compiler options provide a high-level, global approach to optimization. To give you more refinement in optimization, HP C++ provides two pragmas: `OPTIMIZE` and `OPT_LEVEL`.

These pragmas must appear outside any function and they apply for the remainder of the file or until superseded by another pragma. For these pragmas to work, the source program *must* be compiled with one of the options listed in Table 4-2. Otherwise the pragmas are ignored.

4

Pragma OPTIMIZE

The `OPTIMIZE` pragma turns on or off level 2 and 3 optimization. (It does not turn off level 1 optimization.) It is useful for turning off level 2 and 3 optimization in sections of a source program that may cause the optimizer difficulties.

Syntax.

```
#pragma OPTIMIZE { ON  }
                 { OFF }
```

You should specify either `ON` or `OFF`. If you do not, the compiler generates a warning and assumes an `ON` setting.

Examples.

```
#pragma OPTIMIZE ON     Restores original optimization level.
#pragma OPTIMIZE OFF    Sets optimization level to 1.
```

Pragma OPT_LEVEL

The `OPT_LEVEL` pragma directs the compiler to change the current optimization level to level 1, 2, or 3. It is useful for switching from one level to another within a source program.

You cannot use this pragma to raise the optimization level beyond the original level set by the option you used on the `CC` command line. The compiler issues a warning if you attempt to raise the original optimization level.

Syntax.

```
#pragma OPT_LEVEL { 1 }
                  { 2 }
                  { 3 }
```

You should specify an optimization level. If you do not, the compiler generates a warning and assumes level 1.

Examples.

```
#pragma OPT_LEVEL 1    Changes to level 1 optimization.
#pragma OPT_LEVEL 2    Changes to level 2 optimization.
#pragma OPT_LEVEL 3    Changes to level 3 optimization.
```


Optimizing Entire Files

Optimize an entire file by using the appropriate option to the CC command when recompiling your program. You can set the environment variable CXXOPTS with this option as shown below in *sh* notation:

```
CXXOPTS=-O
export CXXOPTS
```

With the CXXOPTS variable set as above, the following command compiles the files sub1.C, app1.C, and mod1.C with level 2 optimization:

```
CC sub1.C app1.C mod1.C
```

Regardless of whether or not the CXXOPTS variable is set, the following example compiles the file app1.C with level 2 optimizations:

```
CC -O app1.C
```

For more information, see the -O and +O options and the CXXOPTS environment variable in Chapter 3.

Optimizing Individual Functions

Use the pragma OPTIMIZE ON before a function definition and OPTIMIZE OFF after the function to request optimization for that function only. You must use either of the -O or +O options for these pragmas to take effect. Otherwise the pragmas are ignored.

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 300/400 and 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.

5

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
```

5

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"`. Refer to "Messages Generated by the Linker" in Chapter 6, "Errors and Diagnostic Messages," for more information on name mangling.
- 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

5 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. On the HP 9000 Series 300/400, FORTRAN 77 passes character lengths by value at the end of the parameter list.

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

5

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.

On the HP 9000 Series 300/400, HP FORTRAN 77 and HP C++ do share a common definition of TRUE (nonzero) and FALSE (0).

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.

On the HP 9000 Series 300/400, 2 bytes are allocated for a boolean and any nonzero value represents TRUE and 0 represents FALSE. On the HP 9000 Series 300/400, HP C++ and HP Pascal do share a common definition of TRUE and 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.

Note On HP 9000 Series 300/400 systems, if you use Pascal I/O, the Pascal heap, or error recovery features of the language, specific run-time setup code is required. Refer to the *HP Pascal Language Reference Manual* for more information.

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 -lnlsstubs -lisamstub
```

On the HP 9000 Series 300/400, the HP FORTRAN 77 run-time library is linked in by adding the arguments

```
-lI077 -lF77
```

to the CC command line.

On the HP 9000 Series 300/400, the Pascal run-time library is linked in by adding the following argument:

5

```
-lpc
```

6

Errors and Diagnostic Messages

This chapter gives you information and guidelines to assist you in understanding the diagnostic messages generated by the HP C++ compiling system. The chapter is organized according to the components of the compiling system that generate messages: `Cpp` (the preprocessor), `cfront` (the C++ compiler/translator), `cc` (the C compiler—used in translator mode only), and `ld` (the linker). The second part of the chapter describes selected diagnostic messages generated by HP C++.

Note A single mistake in your code can sometimes generate several error messages. Usually it's a good idea to correct just the first reported error for each group of closely related line numbers and recompile. That way you can make sure that the remaining errors are truly errors.

Sometimes, however, if the first error message says only "syntax error," the following messages can give more information on the specific problem.

Messages Generated by the Preprocessor

The HP C++ preprocessor generates messages according to the following format:

filename: line_number : error_message

For example, the following message indicates that an error was generated on line 2 of the source file named `test.C`. The preprocessor was unable to execute a directive to include the `header.h` file because it did not find the file.

```
test.C: 2 : Unable to find include file 'header.h'
```

Messages Generated by the HP C++ Compiler

The C++ compiler generates messages in the following format:

```
CC: "filename", line line_number: message_type: message (message_number)
```

Here's an example:

```
CC: "test.C", line 19: error:  argument type expected for foo () (1376)
```

The compiler generates five different types of diagnostic messages:

- **Warnings** alert you to possible problems with the program being processed.
- **Errors** indicate that an error was found in the program being processed. No object file is created.
- **Fatal errors** indicate an error of sufficient severity that the compiler cannot continue processing. No object file is created.
- **Internal errors** indicate an error internal to the compiler. These errors should be reported to your HP support representative.
- **Not implemented** messages indicate that the specified functionality is not available. No object file is created. However, the compiler does attempt to examine the rest of the program.

Messages Generated by the HP C Compiler

When you use translator mode, your HP C++ source code is translated to HP C source code, which is then compiled by the HP C compiler. The HP C compiler reports errors and warnings.

An *error* indicates a problem severe enough to prevent the compiler from creating an executable object file.

A *warning* is less severe than an error. A warning does not prevent the compiler from creating an executable object file. The warning message tells you

about a possible ambiguity in your program for which the compiler believes it can generate the correct code.

Since the HP C++ compiling system is designed to generate error-free C code that is compiled by the HP C compiler, your C++ code should not generate any error messages from the HP C compiler. Your code may, however, generate warnings.

If you *do* get an error message from the HP C compiler, then there is an error in the HP C++ compiling system, and you should contact your HP customer support representative.

Messages Generated by the Linker

The error messages you receive from `cfront` or from the HP-UX link editor, `ld`, are typically the result of unresolved global variables or functions, which are, in turn, the result of references to undefined globals. Output from the linker is processed by `c++filt` before it is sent to `stderr`. The following are examples of messages from the linker:

Example 1

```
/bin/ld:  Unsatisfied symbols:
var1 (data)
```

Example 2

```
/bin/ld:  Unsatisfied symbols:
car::show(int,int,float)       (code)
```

In example 1, variable `var1` was declared and referenced in the program, but its definition was not provided.

In example 2, member function `show` of class `car` was declared and referenced in the program, but the definition or body of the function was not provided.

In translator mode, when `cfront` generates HP C code from your C++ source files, it changes C++ identifiers to unambiguous equivalent C identifiers and resolves the scope of variables, overloaded operators, and overloaded functions. This process is called **name mangling**. Some names are also mangled in compiler mode. When the names are displayed in linker messages, the program

`c++filt` changes the modified C identifiers back to the original C++ names. This process is called **name demangling.**

Since references to global variables are always unambiguous and the identifier can be resolved by standard rules of scope, name mangling is not needed for such references. This is not true for overloaded functions and overloaded operators.

List of Messages

The rest of this chapter describes selected HP C++ diagnostic messages. The messages are arranged according to message number. In each case the actual message is shown along with an explanation of its cause. Text in *italics* within a message represents a value supplied by the compiler reflecting your particular source code.

105 `WARNING  statement after break not reached`

You put a statement after a `break` statement that can never be executed. Move either the `break` statement or the statement following the `break` statement.

```
void main(){
int i, j[3], k[3];

for (i=0;i<=2;i++)
  { j[i]=i;
    break;
    k[i]=1;  // This statement cannot be reached.
  }
}
```

Compiling the above example gives the following message:

```
CC: "test.C", line 7: warning: statement after break not
reached (105)
```

106 WARNING result of *token* expression not used

You created an entity but never used it. For instance, the following simple program calls a constructor to initialize an unnamed object of type `circle`, but the program never uses the object.

```
class circle
{
private: int radius;
public:  circle(int radius) { radius=radius; }
                    //constructor
};
main()
{
   circle(10);      // constructor call
}
```

Compiling the above example gives the following message:

```
CC: "test.C", line 9: warning: result of constructor call
expression not used (106)
```

116 **WARNING** *variable* **used but not set**

You used *variable* without initializing it. If *variable* is a class object, you may have forgotten to define a constructor in its class definition. For instance:

```
#include <iostream.h>
class A
{
   public: int x;
};
main()
{
   A B;  // WRONG -- warning 116:  B used but not set
   int y = 2 * B.x;
   cout << "y is " << y << "\n";
}
```

In this case, you can correct the error by adding a constructor for `A` that initializes `A::x`:

```
class A
{
   public: int x;
   A() { x = 0; }   // RIGHT -- corrects the error
};
```

Another possibility is that you declared an identifier as a pointer to a class object, but forgot to initialize it with the **new** operator.

128 **WARNING** *class_name* **has** *destructor_name* **but no constructor**

You defined a destructor for *class_name*, but you didn't define a *class_name* constructor. It is generally wise to define at least a default constructor, especially if you define a destructor. Otherwise you risk run-time errors resulting from an uninitialized object state.

133 ERROR address of bound function (try using '*type_name* ::*' for pointer type and '&*type_name* ::*func_name*' for address)

You cast a member function for a particular object to a pointer to a function. For example:

```
class S {
public:
    int f();
} s;
typedef int (*PF)();
PF pf = (PF) &s.f;
```

This code generates the following message:

```
CC: "test.C", line 6: error: address of bound function (try
using 'S ::*' for pointer type and '&S ::f ' for address) (133)
```

At version 2.1 of HP C++, this kind of cast was a legal extension to C++. As of version 3.0, this extension is illegal. You should change your code to something like the following:

```
typedef int (S::*PF)();
PF pf = (PF) &S::f;
```

For more information, see sections r.5.4 and r.18.3.4 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

135 WARNING int assigned to enum *type_name* (anachronism)

You assigned an integer value to an enum variable. At version 2.1 of HP C++, such an assignment was a legal extension to C++. At version 3.0 of HP C++, this extension is illegal and causes error number 1213. You should remove such assignments from your code.

For more information, see sections r.7.2 and r.18.3 of the reference manual section of *The C++ Programming Language*.

138 **WARNING** *variable* **used before set**

You used *variable* before initializing it. If *variable* is a class object, you probably did not define a constructor in its class definition. For instance:

```
class A
{
public: int x;
};

main()
{
 A B;
 B.x = 2* B.x;  // WRONG -- warning 138: B used before set
 B.x =1;
}
```

In this case, you can correct the error by adding an A constructor that initializes A::x:

```
class A
{
public: int x;
        A() { x=0; }   // RIGHT -- corrects the error
};
```

142 **WARNING** argument *num*: *type1* passed as *type2*

HP C++ converted an actual function argument to the type required by the function definition. As a result information may have been lost or results could be inaccurate. For instance, a double converted to an int probably gives an inaccurate value.

150 `WARNING '.' used for qualification, please use '::' (anachronism)`

You used the dot (`.`) operator instead of the scope operator (`::`) after a class name, as in the following example:

```
class S {
    int f();
};
int S.f() { return 0; }
```

This was a legal extension at version 2.1 of HP C++, but it became illegal at version 3.0 and generates error number 1671. You should remove such uses of the dot operator from your code.

For more information, see section r.5.1 of the reference manual section of *The C++ Programming Language*.

166 `WARNING assignment to this (anachronism)`

You made an assignment to the `this` pointer. Assignment to the `this` pointer within a class member function definition is a legal extension at versions 2.1 and 3.0 of HP C++, unless you are using exception handling in which case assignment to `this` is an error. Assignment to `this` will be illegal at the next major release of C++. You should remove such assignments from your code.

For more information, see section r.18.3.3 of the reference manual section of *The C++ Programming Language*.

169 `WARNING` *function* `too complex for inlining`

You specified that a function be inlined, but the compiler did not inline the function because it contained a construct that was too complicated. The only way to make the function inline is to simplify it.

218 `WARNING operator new() first argument should be size_t (anachronism)`

You declared your own operator `new()` with a first argument of a type other than `size_t`. Declaring a first argument of a type other than `size_t` was a legal extension at version 2.1 of HP C++, but it is illegal at version 3.0 and generates error number 1131. You should remove such declarations from your code.

For more information, see section r.12.5 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

222 `WARNING` *function* `too large for inlining`

You specified that a function be inlined, but the compiler did not inline the function because it was too large. The only way to make the function inline is to shorten it.

223 `WARNING operator delete()'s 2nd argument should be a size_t (anachronism)`

6

You declared your own operator `delete()` with a second argument of a type other than `size_t`. Declaring a second argument of a type other than `size_t` was a legal extension at version 2.1 of HP C++, but is illegal at version 3.0 and generates error 1137. You should change such declarations to use `size_t`.

For more information, see section r.12.5 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

225 WARNING *type_name* occurs at outer and nested local class scope; using typedef *class_name*::*type_name* (anachronism)

You used a type name that was declared both globally and within a local nested class. For example:

```
typedef int T;          // T has global scope
void f()
{
    struct Nested {
        typedef char T;   // T is local to Nested
    };
    T var;          // At 2.1 this is:  Nested::T (a char)
}                   // At 3.0 this is:  ::T (an int)
```

At version 2.1 and previous versions of HP C++, this conflict was resolved in favor of the local class scope, so that `var` is declared as a `char`. However, at version 3.0 of HP C++ such references are resolved using normal scoping rules, so `var` is an `int`. If you receive this warning using version 2.1 or earlier and do not change your code, your program may behave differently at version 3.0. You should change your code to explicitly qualify such references.

For more information, see section r.9.7 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

226 `WARNING identifier, accessed within nested class` *class_1*`, is visible both globally and within enclosing class` *class_2* `-- using ::`*identifier* `(anachronism)`

In a member function of a nested class, you referenced an identifier that is declared both globally and locally. At previous releases of HP C++, nested classes had global scope. Therefore, if a member function of a nested class accessed an identifier that was both global and local to the enclosing class, the member function always accessed the identifier with global scope. At Release 2.1, the nested class is within the scope of its enclosing class, so accessing an identifier that is both global and local to the enclosing class is ambiguous unless you qualify the name of the identifier. For example:

```
int i;                    // i has global scope
struct S {
    static int i;         // i is local to S
    struct Nested {
         static int f()
         { return i; } // At 2.1 this does: "return S::i;"
                       // At 3.0 this does: "return ::i;"
    };
};
```

6

At version 2.1 of HP C++, the compiler assumed that you intended code like that above to have the same meaning it did at previous releases. That is, `i` referred to the global variable, `::i`. However, at version 3.0 of HP C++ such references are resolved using normal scoping rules, so `i` refers to the local `static int i`. If you receive this warning using version 2.1 or earlier and do not change your code, your program may behave differently at version 3.0. You should change your code to explicitly qualify such references.

For more information, see section r.9.7 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

235 `WARNING initializer for non-const reference not an lvalue (anachronism)`

You initialized a non-const reference with a value that is not an lvalue. At version 3.0 of HP C++, it is a legal extension to use a value that is not an lvalue to initialize a non-const reference. For example:

```
void main(){
int& r = 5;      // legal at versions 2.1 and 3.0
}
```

This extension will become illegal at the next major release of HP C++. You should remove such declarations from your code.

For more information, see section r.8.4.3 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*. For a definition of lvalue, see *The C++ Programming Language*.

238 `WARNING nested` *type_name* `as return type for non-inline member function, use` *class_name*`::`*type_name func_name*`{} in definition (anachronism)`

You defined a type within a class definition, but you did not qualify the type name when you used it as the return type for a member function that you defined outside the enclosing class definition. Using the unqualified type name was a legal extension at version 2.1 of HP C++, but became illegal at version 3.0. You should remove such references from your code.

241 `WARNING out-of-line copy of` *function* `created`

You specified that a function be inlined, but the compiler did not inline the function because it was too large, too complicated, or it was a virtual function. The compiler created an out-of-line copy of the function instead.

249 WARNING use *class_1*::to access nested *type_name class_2* (anachronism)

You defined a type within a class definition, but you did not qualify the type name when you referred to it outside the enclosing class definition. Using an unqualified nested type name outside its enclosing class definition was a legal extension at version 2.1 of HP C++, but it became illegal at version 3.0. You should change such references to explicitly provide the enclosing class. For example:

```
class outside { ...
public:
    class inside { ...
    };
};

void main()
{
   outside h1;
   inside i1;        // WRONG--Type inside is
                     // nested within class outside.
   outside::inside i2;     // RIGHT
}
```

For more information, see sections r.9.7 and r.18.3.5 of the reference manual section of *The C++ Programming Language*.

250 WARNING virtual function *name* cannot be inlined

You specified that a virtual function be inlined, but the compiler could not inline the function because it is virtual. The compiler created an out-of-line copy of the function instead.

273 WARNING *class_name* has *destructor_name* but no constructor

You defined a destructor for *class_name*, but you didn't define a *class_name* constructor. It is generally wise to define at least a default constructor, especially if you define a destructor. Otherwise you risk run-time errors resulting from an uninitialized object state.

900 **WARNING extra subprocess specifications are ignored**

You specified **x** (which means all subprocesses) with the **-t** option, but you also specified other subprocesses. The **CC** driver ignores the extra ones.

901 **WARNING unknown option: '***option_name***'**

You probably made a typing mistake and used an unknown option. For instance, if you accidentally use **-V** instead of **-v** you get the following message:

```
CC: warning: unknown option: '-V' (901)
```

See Chapter 3 for information about the options that are supported.

1001 **ERROR** *identifier_name* **declared as** *type1* **and** *type2*

You declared *identifier_name* to be an object of two different types.

1002 **ERROR** *identifier_name* **declared as identifier and typedef**

You declared an identifier and a typedef with the same name.

1013 **WARNING 'overload' used (anachronism)**

Your program contains an **overload** statement. Use of the obsolete keyword **overload** is a legal extension at version 3.0 of HP C++, but it will be illegal at the next major release. You should remove **overload** statements from your code.

For more information, see sections r.2.4 and r.18.3 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

1017 **ERROR two declarations of** *identifier_name*; **types:** *type1* **and** *type2*

You declared two entities that have the same name but different types. Fix this by changing one of the names or one of the types.

1034 ERROR two definitions of *name*

You defined *name* twice. For instance, the following code fragment shows a function that was defined inline when the function was declared in a header file and then defined again in the implementation file. Notice that you get this error even if the definitions are exactly the same.

```
// This is the account.h header file.

class account
{
private:
    double balance;
public:
    void deposit(double amount) { balance += amount; }
};

// This is the account.C file.

#include "account.h"
void account::deposit(double amount)
    { balance += amount; }
main()
{}
```

The preceding code generated this error message:

```
CC: "account.C", line 4: error: two definitions of
account::deposit() (1034)
```

1039 ERROR array of class *class_name* that does not have a constructor taking no arguments

If you want to declare an array of objects of a class, the class must have either a constructor that does not require an argument list or no constructor at all. For instance, the account class has a constructor that takes one argument in the following code fragment:

```
class account
{
      :
    account(char* c)     // constructor
       { name = new char [strlen(c) +1];
         strcpy(name,c);
         balance = rate = 0; }
};
      :
account customers[5];    // WRONG
```

1046 ERROR uninitialized reference *name*

You failed to initialize *name*, which is a reference variable:

```
int& ri;                 // WRONG
```

You declare a non-static reference variable by appending the address-of operator (&) to the type name and including an initializer:

```
const int& ri=0;         // RIGHT
int i = 3;
int& ri2 = i;            // RIGHT
```

Static reference variables must be initialized with an lvalue (location value) rather than an rvalue (data value):

```
int i = 3;
static int& ri3 = i;     // RIGHT
```

For the definition of lvalue, see *The C++ Programming Language*.

1049 ERROR bad initializer type *type1* for *name* (*type2* expected)

You initialized an object with the wrong type. For instance, the `show_radius()` function returns a value of type `void`, but the `x` variable must be initialized with an `int` type value:

```
class circle
{
   :
   void show_radius(){ }   // member function
};
main()
{
   circle C = circle(1);
   int x =C.show_radius(); // WRONG
}
```

The preceding code generates the following message:

```
CC: "test.C", line 5: error: bad initializer type void for x
(int expected) (1049)
```

6 1058 ERROR unexpected return value

You used a `return` statement to return a value from a function declared as `void` or from a constructor or a destructor. Constructors and destructors do not return a value.

1064 ERROR case not in switch

You tried to use a case statement outside of a switch statement. It is possible that you just need to delimit the case statements with braces ({ }):

```
switch (answer)
      case 'A': case 'a': cout<<"\nyou selected A\n"; break;
      case 'B': case 'b': cout<<"\nyou selected B\n"; break;
      case 'C': case 'c': cout<<"\nyou selected C\n";
          // WRONG

switch (answer)
{
      case 'A': case 'a': cout<<"\nyou selected A\n"; break;
      case 'B': case 'b': cout<<"\nyou selected B\n"; break;
      case 'C': case 'c': cout<<"\nyou selected C\n";
}         // RIGHT
```

1102 ERROR two different return value types for *function_name*: *type1* and *type2*

You declared *function_name* with two different types for the return values. For instance, in the following code fragment the first declaration of the fun() function indicates a return value of type int by default, but fun() is also defined to return a void:

```
fun() {};
void fun() {};
```

The code in this example generates the following message:

```
CC: "test.C", line 5: error: two different return value types
for fun():  int  and  void  (1102)
```

1113 ERROR class *name* defined twice

You defined the same class twice. For instance:

```
class A
{
   int data;
};
class A
{
   int data;
};          // WRONG
```

1120 ERROR two initializers for *constructor_function* argument *arg_name*

You probably initialized a constructor function argument both in the definition and in the declaration of the function. You may initialize constructor function arguments in either the declaration or the definition, but not both.

1124 ERROR *storage_class* specified for qualified name *function_name*

You may have used the `static` specifier in the definition of a static member function. You may use the `static` specifier only in the declaration. For example:

```
static int Robot::busy(short latit, short longit) // WRONG
{...}
```

This code generates the following message:

```
CC: "test.C", line 5: error: static specified for qualified
name busy() (1124)
```

1131 ERROR `operator new() requires a first argument of type size_t`

You declared your own operator `new()` with a first argument of a type other than `size_t`. Declaring a first argument of a type other than `size_t` was a legal extension at version 2.1 of HP C++, but it is illegal at version 3.0. You should change such declarations to use `size_t`.

For more information, see section r.12.5 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

1137 ERROR `operator delete()'s 2nd argument must be a size_t`

You declared your own operator `delete()` with a second argument of a type other than `size_t`. Declaring a second argument of a type other than `size_t` was a legal extension at version 2.1 of HP C++, but is illegal at version 3.0. You should change such declarations to use `size_t`.

For more information, see section r.12.5 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

1143 ERROR *mem_func_name* `type mismatch:` *mem_func_arg_type1* and *mem_func_arg_type2*

Most likely, an argument type in the declaration of a class member function (*mem_func_arg_type1*) does not match the argument type in its definition (*mem_func_arg_type2*).

1147 ERROR `non-class virtual` *function_name*

You specified *function_name* as `virtual` but *function_name* is not a class member function.

1150 ERROR *operator_name* `must take 1 or 2 arguments`

You probably used the wrong number of arguments when you overloaded an operator. A non-member function requires one argument if it's a unary operator and two arguments if it's a binary operator. For instance:

```
class A { };
int operator+ (A one, A two, A three);  // WRONG
int operator+ (A one, A two);           // RIGHT
```

1151 ERROR *operator_name* `must take 0 or 1 arguments`

You probably used the wrong number of arguments when you overloaded an operator. A member function requires no arguments if it's a unary operator and one argument if it's a binary operator. (An overloaded unary operator which is also a member function uses this as its operand. A binary operator uses this as the left operand.) For instance:

```
class A {
int operator+(A one, A two);      // WRONG
int operator+(A one);             // RIGHT
};
```

1169 ERROR *identifier_name* `redefined: identifier and typedef`

You declared an identifier and a typedef with the same name. For instance:

```
int N;
typedef N;
```

Compiling the above example gives the following message:

```
CC: "test.C", line 2: error: N redefined: identifier and
typedef (1169)
```

1170 ERROR `initializer for member` *class_member_name*

You tried to initialize *class_member_name* when you declared it. You must use a constructor to initialize class data members. For instance:

```
class A
{
private: int one,two,three=0;      // WRONG
};

class A
{
private: int one,two,three;
public:
 A() { one=two=three=0; }          // RIGHT
};
```

1185 ERROR `'this' used in non-class context`

You referred to the `this` pointer, but you are not defining a class member function. You may have forgotten to use the scope operator (`::`) in the member function definition.

1193 ERROR `non-object .`*member_name*

You may have used the dot (`.`) operator instead of the right-arrow (`->`) operator when using a pointer to refer to a class member, *member_name*. As a result, it appears that you are trying to use *member_name* as if it were a member of an object that doesn't exist. For instance:

```
class A {
public: int a;
};
main(){
A* Ap;
Ap.a=2;  // WRONG -- there's no Ap object; Ap is a pointer
Ap->a=2; // RIGHT
}
```

1196 ERROR pointer to member expected in .* expression: *type*

You used a pointer to a type when you should have used a pointer to a class member function. For instance, suppose you make the following declarations:

```
class course { ... };
void (course ::* pmfc)(); // pmfc is a pointer to a member
                          // function of a course object
course math;              // math is a course object
char * c;
```

Then the following function call is illegal:

```
(math.*c)();              // WRONG
```

You probably meant something like this:

```
(math.*pmfc)();           // RIGHT
```

1197 ERROR bad object type in .* expression: *type* (*class_type** expected)

6 You used *type* when you should have used *class_type*. For instance, suppose you make the following declarations for pmfc, c, and math:

```
class course { ... };
   // declare pmfc as a pointer to a member
   // function of a course object
void (course ::* pmfc)();
char c;
course math;
```

Then the following function call is illegal:

```
(c.*pmfc)();     // WRONG--c is not a class type
                 // error 1197: bad object type in .*
                 // expression:  char * (course * expected)
```

But the following is correct:

```
(math.*pmfc)(); // RIGHT--math is a class type
```

1199 ERROR subscript expression missing

You left out an expression required inside square brackets, probably in referencing an array element. For instance:

```
A[]=1;  // WRONG
A[0]=1; // RIGHT
```

1213 ERROR bad assignment type: *type1* = *type2*

You attempted to assign a value of the wrong type to a variable. For instance, the following code incorrectly assigns a `void` function return type (*type2*) to an `int` type variable (*type1*):

```
class circle
{
   :
   void show_radius(){ }
};
main()
{
   :
   int x;
   x=C.show_radius();     // WRONG
}
```

The preceding code generates the following error message:

```
CC: "test.C", line 5: error: bad assignment type: int = void
(1213)
```

1237 ERROR bad operand types *type1 type2* for *operator*

You used the wrong types of variables with *operator*. If you overloaded *operator*, check your definition of the overloaded operator.

1261 ERROR bad argument list for *function_name* (no match against any *function_name*)

You failed to supply a set of arguments for *function_name* that matches any of the definitions for *function_name*.

1262 ERROR object or pointer missing for *function_name* of type *type*

You may have treated a non-static member function as if it were static, instead of referring to the particular instance of the function. For instance, the following code declares a non-static member function but then calls it using the scope operator, which is only allowed with static member functions:

```
class Robot {
    int unit;
public:
    void energize();
    Robot() { unit = 0;}
};
void Robot::energize() { unit++; }
int main(void)
{
    Robot *robot_instance = new Robot;

    Robot::energize();              // WRONG
    robot_instance->energize();     // RIGHT
}
```

This code generates the following error message:

```
CC: "test.C", line 12: error: object or pointer missing for
Robot::energize() of type void Robot::() (1262)
```

1263 WARNING non-const member function *function_name* called for const object (anachronism)

You called a non-const member function for a const object. You can call a const member function for both const and non-const objects, but you can call a non-const member function only for non-const objects. For example:

```
class S {
public:
    int f();          // non-const member function
};
extern const S s;     // const object
int i = s.f();
```

This code generates the following error message:

```
CC: "test.C", line 6: warning: non-const member function
S::f() called for const object (anachronism) (1263)
```

This is a legal extension at version 2.1 of HP C++, but it is illegal at version 3.0. To call a member function for a const object, declare the member function const as follows:

```
int f() const;
```

For more information, see section r.9.3.1 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

1264 ERROR bad argument *num* type for *function_name*: *type1* (*type2* expected)

You used *type1* as the argument type for the *num* argument of the *function_name* function when you should have used *type2*. For instance, the following `fill_array` function requires a `const int` as the type for its first argument, but instead the function is called with a character array element:

```
void fill_array(const int i, char c)
{
   :
}
main()
{
   char A[7];
   fill_array(A+1,'B');   // WRONG
   fill_array(*A+1,'B');  // RIGHT
}
```

This program generates the following error message:

```
CC: "test.C", line 12: error: bad argument 1 type for
fill_array(): char [7] (const int expected) (1264)
```

1265 ERROR unexpected *num* argument for *function_name*

You passed a function argument where none was required.

1270 ERROR bad initializer type: *type1* (*type2* expected)

You used a *type1* value to initialize a *type2* object. For instance:

```
char& y = "A";  // WRONG
```

The above example generates the following message:

```
CC: "test.C", line 1: error: bad initializer type: char [2]
(char & expected) (1270)
```

In this case, you can correct the error as shown below:

```
const char& y = 'A';  // RIGHT
```

1273 ERROR cannot make a *class_object* from a *type1*

You tried to convert a *type1* variable to a *class_object* without defining the necessary type conversion. You need to add a *class_object* constructor with a single argument of *type1* to specify a conversion from *type1* to *class_object*. For example:

```
class C{ ... };
class D{ ... };

void f(C);   // f() expects a parameter of type C

void main(){
C c;
D d;
f(d);        // WRONG: passing a D where a C is expected
}
```

Compiling the above example gives the following message:

```
CC: "test.C", line 8: error: cannot make a C from a D (1732)
```

1285 ERROR *name* undefined

You used a name without defining it. Many C++ programs use functions that are defined in header files. To use such functions you must be sure to include the necessary header files in your source code. For instance, the following message resulted from the omission of a `#include <iostream.h>` directive, because the cout function is defined in the `iostream.h` file.

```
CC: "test.C", line 6: error: cout undefined (1285)
```

1286 ERROR undefined function *function_name* called

You called *function_name*, but you didn't define it.

1287 ERROR member *class_member* undefined

You tried to use an undefined class member. For instance, in the following simple program, the N class definition does not contain a b member, so the assignment of a value to the Nobject.b member fails.

```
class N {
public: int a;
};

main()
{
   N Nobject;
   Nobject.b =3;     // WRONG
}
```

This program generates the following message:

```
CC: "test.C", line 8: error: member b undefined (1287)
```

1291 ERROR object or object pointer missing for member *class_member*

You may be trying to access a non-static data member within a static member function, using the scope operator. Static member functions may only access static data members. For example:

```
class Robot {
    int unit;  // this must be declared static
public:
    static void energize();
    Robot() { unit = 0;}
};
void Robot::energize()
{
    Robot::unit++;  // WRONG
}
```

This code generates the following error message:

```
CC: "test.C", line 9: error: object or object pointer missing
for member unit (1291)
```

1293 ERROR object or object pointer missing for *class_member*

You may be trying to access a non-static data member within a static member function without using the scope operator. Static member functions can only access static data members. For example:

```
class Robot {
    int unit;      // this must be declared static
public:
    static void energize();
    Robot() { unit = 0;}
};
void Robot::energize()
{
    unit++;        // WRONG
}
```

This code generates the following error message:

```
CC: "test.C", line 9: error: object or object pointer missing
for Robot::unit (1293)
```

1299 ERROR *function_name* cannot access *class_member*: *access_specifier* member

You tried to use a class member to which you do not have access. You may have forgotten to use the scope operator (::) in a member function definition.

1310 ERROR *class_object_name* :: *class_object_name* is not a type name

You confused the type name for a class you created with the name of a particular object of the class. When you want to refer to a member of a particular object, you must use the name of that particular object, rather than the type name.

For instance, the following code declares a class with the type name N and a particular object of that class named Nobject. It assigns a value to the a member of the Nobject:

```
class N {
public: int a;
};

main()
{
   N Nobject;
   Nobject::a=2; // WRONG- error: Nobject :: Nobject
                 // is not a type name (1310)
   Nobject.a=2;  // RIGHT
}
```

1311 ERROR *tagname* is hidden: use struct *tagname name*

You used the same name for an object and a structure tag name, and you failed to use the struct keyword preceding the tag name of the structure. You must explicitly precede the tag name with the struct keyword.

1314 ERROR *type_name variable*: *type_name* is not a type name

You used something as a type that either is not actually a type or is not directly accessible. If it is not directly accessible, change it to provide the correct access syntax. For example:

```
class outside { ...
public:
    class inside { ...
    };
};

void main()
{
   outside h1;
   inside i1;        // WRONG--Type inside is
                     // nested within class outside.
   outside::inside i2;     // RIGHT
}
```

For more information, see sections r.9.7 and r.18.3.5 of the reference manual section of *The C++ Programming Language*.

1324 ERROR '}' missing at end of input

You probably forgot to end your program with a right brace. If it seems as if the right brace is there, check for an earlier error that could have caused this to be a spurious message. For instance, there might be a missing right brace earlier in the program, so the left and right braces are not balanced.

1326 ERROR illegal character '*character*' (ignored)

You used *character* illegally. You might have omitted double quotation marks necessary to enclose a string. For instance:

```
cout << "y = << y << "\n";    // WRONG- illegal character '\'
cout << "y = " << y << "\n"; // RIGHT- add quotation marks
```

Although HP C++ ignores this error and continues to process your source code, you might get several additional errors as a result of this one mistake.

1354 ERROR *identifier* redefined: previous: *type1* now: *type2*

You defined *identifier* twice: first with *type1* and then with *type2*. You have to change one of the definitions.

1367 ERROR *type* defined as return type for *function_name* (did you forget a ';' after '}' ?)

You probably forgot to put a semicolon (;) after the right brace (}) at the end of a class definition. For instance:

```
class circle
{
   :
}                    // WRONG--missing semicolon
main()
{
   :
}
```

1368 ERROR enum *variable_name* defined as return type for *function_name* (did you forget a ';'?)

You probably forgot to put a semicolon (;) after the right brace (}) enclosing the body of an enum definition. For instance:

```
enum beatles
   {john,paul,george,ringo} // WRONG--missing semi-colon
main()
{}
```

The preceding code generates the following message:

```
CC: "test.C", line 3: error: enum beatles defined as return
type for main() (did you forget a ';'?) (1368)
```

Notice that the error is generated for line 3 of the program, even though the missing semicolon belongs on line 2.

1369 ERROR `body of non-function` *function_name*

You probably omitted the parentheses in the definition of a function. As a result, the compiler reports that you haven't declared a function for which you defined the body. The line number refers to the line containing the right brace (}) enclosing the function body. For instance:

```
main                    // WRONG--missing parentheses
{}
```

The preceding code generates the following error message:

```
CC: "test.C", line 2: error: body of non-function main (1369)
```

To correct this error, add the parentheses as shown:

```
main()                  // RIGHT
{}
```

1376 ERROR `argument type expected for` *function_name*

Function declarations must include types for function arguments. You may have left this off a function declaration. You might also get this error when you intend to call a function rather than declare one. This could happen if an earlier error in your code causes the function call to be interpreted as a declaration.

1386 WARNING old style definition of *function_name* (anachronism)

You defined a function using old-style syntax (that is, without a prototype). For example:

```
int f(i)
int i;
{
    return i;
}
```

This is a legal extension at version 3.0 of HP C++, but it will be illegal at the next major release. You should convert such definitions into function prototypes.

If you compile with the +p option, this message is an error rather than a warning.

For more information, see section r.18.3.1 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

6

1401 ERROR member *array_of_class_name_objects*[] needs initializer (no default constructor for class *class_name*)

You declared an array of objects of type *class_name*, but *class_name* does not have a default constructor. The easiest way to correct this error is to add a default constructor for *class_name*. You can also correct the error by initializing each element of *array_of_class_name_objects*.

For instance, the following code generates an error because the account class does not have a default constructor:

```
class account
{
   :
public:
    account(char* c)       // constructor takes one argument
       {
          :
       }
};
class bank
{
 account customers[5];     // WRONG
 bank() { }
};
```

In this case you can correct the error by adding a default constructor, account(), as a member function for the account class.

1407 ERROR statement or initialized expression not reached: (case label missing?)

You put a statement where it can't be reached inside a `switch` statement. For example:

```
switch (answer)
{
  char * response = "\nyou selected "; // WRONG- this can't be reached

  case 'A': case 'a': cout << response << "A\n"; break;
  case 'B': case 'b': cout << response << "B\n"; break;
  case 'C': case 'c': cout << response << "C\n"; break;
}
```

You can correct the error in this example by placing the statement initializing the `response` variable just before the `switch` statement:

```
char * response = "\nyou selected ";     // RIGHT
switch(answer)
{
   case 'A': case 'a': cout << response << "A\n"; break;
   case 'B': case 'b': cout << response <<" B\n"; break;
   case 'C': case 'c': cout << response <<" C\n"; break;
}
```

1421 ERROR `float` operand for *operator*

You tried to use a `float` type variable with an operator that doesn't work with the `float` type. This error could be the result of a mistake in overloading the operator: either you failed to include a file that contains the definition for the overloaded operator, or you did not define the overloaded operator properly. If the operator is << or >>, you may have failed to include the `iostream.h` header file, which overloads these operators.

1422 ERROR `pointer operand for` *operator*

You tried to use a `pointer` type variable with an operator that doesn't work with the `pointer` type. This error could be the result of a mistake in overloading the operator: either you failed to include a file that contains the definition for the overloaded operator, or you did not define the overloaded operator properly. If the operator is `<<` or `>>`, you probably failed to include the `iostream.h` header file, which overloads these operators.

1424 ERROR `reference operand for` *operator*

You tried to use a reference type variable with an operator that doesn't work with the reference type. This error could be the result of a mistake in overloading the operator: either you failed to include a file that contains the definition for the overloaded operator, or you did not define the overloaded operator properly.

1425 ERROR `function operand for` *operator*

You tried to use a function with an operator that doesn't work with functions as operands. This error could be the result of a mistake in overloading the operator: either you failed to include a file that contains the definition for the overloaded operator, or you did not define the overloaded operator properly.

1439 ERROR `non-pointer dereferenced`

You tried to dereference a non-pointer variable. For instance:

```
int i;
int& ri = i;  // ri is a reference variable
              // not a pointer variable.
int j = *ri;  // WRONG- only pointers can be dereferenced.
int k = ri;   // RIGHT
```

1458 ERROR *function_name*'s definition is nested (did you forget a '}'?)

You probably omitted the right brace (}) required to indicate the end of a block preceding the definition of *function_name*. As a result, the definition for *function_name* appears to be nested inside the block. For instance:

```
void fun1()
{               // WRONG--no matching right brace
void fun2()
{}
```

The above example generates the following message:

```
CC: "test.C", line 3: error: fun2()'s definition is nested
(did you forget a '}'?) (1458)
```

1466 ERROR argument declaration syntax

You made some sort of error in declaring the arguments to a function. For example:

```
int f(int i, j);  // WRONG - should be (int i, int j);
```

This example causes the following error message:

```
CC: "test.C", line 1: error: argument declaration syntax
(1466)
```

1502 ERROR `syntax error`

There are many possible causes for this error. You probably made a syntax error on the line for which the error is reported, so you should first check that line for missing semicolons and other incorrect syntax. If you cannot find a syntax error on that line, check for an earlier semantic error that could have caused a spurious syntax error. For example, a `try` block with no `catch` statements following it causes this error.

1512 ERROR `initializer for non-const reference not an lvalue`

You initialized a non-const reference with a value that is not an lvalue. At version 2.1 and version 3.0 of HP C++, it is a legal extension to use a value that is not an lvalue to initialize a non-const reference (unless you use the `+p` option). For example:

```
void main(){
int& r = 5;      // legal at versions 2.1 and 3.0
}
```

This extension will become illegal at the next major release of HP C++. You should remove such declarations from your code.

For more information, see section r.8.4.3 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*. For a definition of lvalue, see *The C++ Programming Language*.

1522 `WARNING name of base class class_name missing from base class initializer (anachronism)`

You used an old-fashioned method to initialize a base class in a derived class constructor. If a derived class has exactly one immediate base class, you may omit the base class name from the derived class constructor, as follows:

```
class B {
    int b;
public:
    B(int i) { b = i; }
};
class D : public B {
    D(int i) : (i) {}          // should be B(i)
};
```

This is a legal extension at both versions 2.1 and 3.0 of HP C++, but it will be illegal at the next major release of the C++ Language System. You should modify your code to include the base class name.

For more information, see section r.18.3.2 of the reference manual section of *The C++ Programming Language*.

1531 `ERROR non-member operator =()`

You declared `operator=()` globally rather than as a member function. The assignment function `operator=()` must be declared as a nonstatic member function. Declaring it globally was a legal extension at version 2.1 of HP C++, but it is illegal at version 3.0. You should make this operator a class member function.

For more information, see section r.13.4.3 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

1537 `WARNING static member` *member_name* `in local class` *class_name* `(anachronism)`

You declared a static data member in a class that is declared inside a function definition. For example:

```
main() {
class myclass
  {
    public: static int value;
  };
}
```

Compiling the above program gives the following message at version 2.1:

```
CC: "test.C", line 3: warning: static member myclass::value in
local class myclass (anachronism) (1537)
```

Allowing static data members in a local class was a legal extension at version 2.1 of HP C++, but it is illegal at version 3.0. You should remove such declarations from your code.

For more information see sections r.9.4 and r.9.8 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

1549 `WARNING v in '::delete[v]' is redundant; use '::delete[]' instead (anachronism)`

You specified the number of elements in the array you are deleting with the global `::operator delete()`. Earlier implementations of C++ required that you specify the number of elements in the array. Release 3.0 of C++ does not impose this requirement. Specifying the number of elements in an array you are deleting is a legal extension at version 3.0 of HP C++, but it will be illegal at the next major release. You should remove such statements from your code.

For more information, see sections r.5.3.4 and r.18.3 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

1550 `WARNING v in 'delete[v]' is redundant; use 'delete[]' instead (anachronism)`

You specified the number of elements in the array you are deleting with `operator delete()`. Earlier implementations of C++ required that you specify the number of elements in the array. Release 3.0 of HP C++ does not impose this requirement. Specifying the number of elements in an array you are deleting is a legal extension at version 3.0 of HP C++, but it will be illegal at the next major release. You should remove such statements from your code.

For more information, see sections r.5.3.4 and r.18.3 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

1573 `ERROR two declarations of` *identifier_name*`;` *type1* `and` *type2*

You declared two entities that have the same name but different types. For instance:

```
int N;
double N;
```

The above example generates the following message:

```
CC: "test.C", line 2: error: two declarations of ::N; int and
double (1573)
```

You can fix this by changing one of the names or one of the types.

1580 ERROR jump past initializer (did you forget a '{ }'?)

You declared and initialized a variable in a place where it could be jumped over and not executed. For example:

```
void main()
{
        int     i = 7;
        switch(i)
        {
        case 1:
                for(int j = 0; j < 20; j++)  // WRONG
                        ;
                break;
           :
        }
}
```

Initializations that might be jumped past must be contained in an inner block where the entire block is jumped past. For example:

```
void main()
{
        int     i = 7;
        switch(i)
        {
        case 1:
                { for(int j = 0; j < 20; j++)  // RIGHT
                        ;
                  break;
                }
           :
        }
}
```

An alternative is to move the declaration of the variable up so the initialization becomes an assignment. For example, you could move the declaration to immediately before the `switch` statement.

For more information see section r.6.3.4 of the reference manual section of *The C++ Programming Language* and "HP C++ Version 3.0 Detects More Errors" in Chapter 1 of this book.

1587 **WARNING** **static member** *member_name* **in local class** *class_name*

You declared a static data member in a class that is declared inside a function definition. For example:

```
main() {
class myclass
  {
    public: static int value;
  };
}
```

Compiling the above program gives the following message:

```
CC: "test.C", line 3: warning: static member myclass::value in
local class myclass (1587)
```

Allowing static data members in a local class was a legal extension at version 2.1 of HP C++, but it is illegal at version 3.0. You should remove such declarations from your code.

For more information see sections r.9.4 and r.9.8 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

1590 ERROR class *name* defined twice

You defined the same class twice. For instance:

```
class A
{
   int data;
};
class A
{
   int data;
};          // WRONG
```

1627 ERROR pointer to member expected in ->* expression: *type*

You used a pointer to a type when you should have used a pointer to a class member function. For instance, suppose you make the following declarations:

```
class course { .. };
void (course ::* pmfc)(); // pmfc is a pointer to a member
                          // function of a course object
course math;
course* p_course = &math;// p_course is a pointer to
                          // a course object
char * c;
```

Then the following function call is illegal:

```
(p_course->*c)();                 // WRONG
```

You probably meant something like this:

```
(p_course->*pmfc)();              // RIGHT
```

1634 ERROR bad operands for *operator*: *type1 operator type2*

You used the wrong types of variables with *operator*. If you overloaded *operator*, check your definition of the overloaded operator.

1671 ERROR '.' used for qualification, please use '::'

You used the dot (.) operator instead of the scope operator (::) after a class name, as in the following example:

```
class S {
    int f();
};
int S.f() { return 0; }
```

This was a legal extension at version 2.1 of HP C++, but is illegal at version 3.0. You should remove such uses of the dot operator from your code.

For more information, see section r.5.1 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

1675 ERROR *function_name*'s definition is nested (did you forget a '}'?)

You probably omitted the right brace (}) required to indicate the end of a block preceding the definition of *function_name*. As a result, the definition for *function_name* appears to be nested inside the block. For instance:

```
void fun1()
{               // WRONG--no matching right brace
void fun2()
{}
```

The above example generates the following message:

```
CC: "test.C", line 3: error: fun2()'s definition is nested
(did you forget a '}'?) (1675)
```

1692 ERROR *tagname* `is in this scope: use class` *tagname name*

You used the same name for an object and a structure or class tag name, and you failed to use the `struct` or `class` keyword preceding the tag name of the structure or class.

For example, consider the following fragment:

```
void foo(int* x, int* y)
   :
class foo
   :
main()
{
   :
   foo(xp,yp);
   foo S;          // WRONG
   class foo S;    // RIGHT
}
```

The code in this example uses the name `foo` for both a function declaration and a `class` name. You must precede the class name with the `struct` or `class` keyword. If you declare the `S` object in this example without the `struct` keyword, you get a message like this:

```
CC: "test.C", line 9: error: foo is in this scope: use class
foo S (1692)
```

1696 ERROR *identifier* redefined: *type1* and typedef *type2*

You defined *identifier* twice: first with *type1* and then with *type2*. You have to change one of the definitions. For instance:

```
class A {};
typedef char* A;    //  WRONG
```

This example generates the following message:

```
CC: "prog.C", line 2: error: A redefined: struct A and typedef
char * (1696)
```

1724 ERROR non-const member function *function_name* called for const object

You called a non-const member function for a const object. You can call a const member function for both const and non-const objects, but you can call a non-const member function only for non-const objects. For example:

```
class S {
public:
    int f();          // non-const member function
};
extern const S s;     // const object
int i = s.f();        // WRONG
```

This code generates the following error message:

```
CC: "test.C", line 6: error: non-const member function S::f()
called for const object (1724)
```

This is a legal extension at version 2.1 of HP C++, but it is illegal at version 3.0. To call a member function for a const object, declare the member function const as follows:

```
int f() const;
```

For more information, see section r.9.3.1 of the reference manual section of *The C++ Programming Language* and "Future Compatibility Issues" in chapter 4 of the *C++ Language System Release Notes*.

1730 ERROR initializer for global non-const reference not an lvalue

You must use an lvalue (location value) rather than an rvalue (data value) to initialize a global, non-constant reference. For the definition of lvalue, see *The C++ Programming Language*.

For instance, the following code generates an error because 3 is not an lvalue:

```
static int& ref_value = 3;                    // WRONG
```

To avoid this error, either make the static reference const or initialize a variable and use the variable to initialize the static reference:

```
int value = 3;
static int& ref_value = value;                // RIGHT
```

1735 ERROR no match for call:

You failed to supply a set of arguments for a function that matches any of the definitions for the function. For example:

```
void f1(int,int);
void f1(int);
void main(){
f1();
}
```

Compiling the above example gives the following message:

```
CC: "test.C", line 4: error: no match for call: f1 ()  (1735)
CC: "test.C", line 4: choice of f1()s:
 CC: "test.C", line 4:          f1(int , int );
 CC: "test.C", line 4:          f1(int );
```

Either change the call to match one of the existing functions or define a new function to match the call.

1807 ERROR *mem_func_name* type mismatch: *mem_func_arg_type1* and *mem_func_arg_type2*

Most likely, an argument type in the declaration of a class member function (*mem_func_arg_type1*) does not match the argument type in its definition (*mem_func_arg_type2*).

1821 ERROR use +eh for exception handling

You forgot to use the +eh option when compiling. 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.

1900 ERROR cannot dup file descriptor *num* for *name*

The error is in the HP C++ product, not in your code. Please contact either your HP Response Center or your local HP representative.

1901 ERROR cannot exec *name*

You may have gotten this error by specifying a directory as *name* instead of a filename: the -t*x* option requires a filename instead of a directory name when you have just one value for *x*.

1902 ERROR cannot open *filename* for %s

An input or output file cannot be opened, possibly because you do not have permission to do so.

1903 ERROR cannot remove *name*

The error is in HP C++, not in your code. Please contact either your HP Response Center or your local HP representative.

1904 ERROR cannot rename *name* to *name*

The error is in HP C++, not in your code. Please contact either your HP Response Center or your local HP representative.

1905 ERROR cmdline package not initialized

The error is in HP C++, not in your code. Please contact either your HP Response Center or your local HP representative.

1906 ERROR `cannot create new process`

There are too many processes running on your system.

1907 ERROR `no file specified for %s`

The error is in HP C++, not in your code. Please contact either your HP Response Center or your local HP representative.

1908 ERROR `out of memory`

The error is in HP C++, not in your code. Please contact either your HP Response Center or your local HP representative.

1911 ERROR `bad argument for '`*`option_name`*`' option`

You specified an incorrect argument for the *option_name* option or omitted an argument required by *option_name*.

1912 ERROR `illegal argument to '`*`option_name`*`': 0 or 1 expected`

The only legal values for *option_name* are 0 and 1.

1913 ERROR `'`*`filename`*`' does not exist or cannot be read`

6 If it seems like *filename* exists, make sure that you typed *filename* correctly, that you have permission to read it, and that you did not leave the file open for editing.

1914 ERROR `bad form for '`*`option_name`*`' option`

You used an incorrect or incomplete form of *option_name*. For instance, you might have specified `-WP file` when you meant to specify `-WP,file`.

1915 ERROR `cannot overwrite source file '`*`filename`*`'`

You specified an output file that was the same as an input file. For instance, you might have used the `-F` option in combination with the *-.suffix* option, as shown below:

```
CC -F -.c prog.c                    // WRONG
```

One way to avoid this problem is to use a different suffix for the input file:

```
CC -F -.c prog.C                    // RIGHT
```

1917 `ERROR expected subprocess arguments for 'option_name' option`

You probably typed a space between the subprocess required by *option_name* and the required argument list. For instance, if you type `-WP, file` instead of `-WP,file`, you get the following error message:

```
CC: error: expected subprocess arguments for '-W' option
(1917)
```

1918 `ERROR unknown subprocess specification: 'character'`

You used an illegal value for *c* with the `-Wc` or the `-tc` options. See Chapter 3 for more information about these options.

2000-2999 `NOT IMPLEMENTED`

Messages numbered 2000 to 2999 by HP C++ are designated as not implemented by the translator on which HP C++ is based. See the *C++ Language System Product Reference Manual* for more details about these messages.

3000-3999 `INTERNAL/LIMIT`

Messages numbered 3000 to 3999 are internal or limit errors.

An internal error indicates an error in the HP C++ product, not in your code. Please contact either your HP Response Center or your local HP representative.

A limit error means that your code exceeded the limits of the compiling system. If you are unable to work around the limit, please contact either your HP Response Center or your local HP representative.

4200-4201 `Code Generator Internal Errors`

Messages numbered 4200 and 4201 are internal errors related to code generation. An internal error indicates an error in the HP C++ product, not in your code. Please contact either your HP Response Center or your local HP representative.

7

lex: A Lexical Analyzer and Generator

Introduction

The `lex` command is a program generator designed for lexical processing of character input streams. It accepts a high-level problem oriented specification for character string matching and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to `lex`. The `lex` generated code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings, program sections provided by the user are executed. The `lex` source file associates the regular expressions and the program fragments. As each expression appears in the input to the program generated by `lex`, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high-level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

lex: A Lexical Analyzer and Generator

The `lex` command is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." At present, the only host language is C. Just as general purpose languages can produce code to run on different computer hardware, `lex` can generate code in different host languages. The host language is used for the output code generated by `lex` and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes `lex` adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. The `lex` command itself exists on HP-UX, but the code generated by `lex` may be taken anywhere the appropriate compilers exist.

The `lex` command turns the user's expressions and actions (called *source*) into the host general-purpose language. The generated program is named `yylex`, and recognizes expressions in a stream (called *input*). The `yylex` command performs the specified actions for each expression as it is detected.

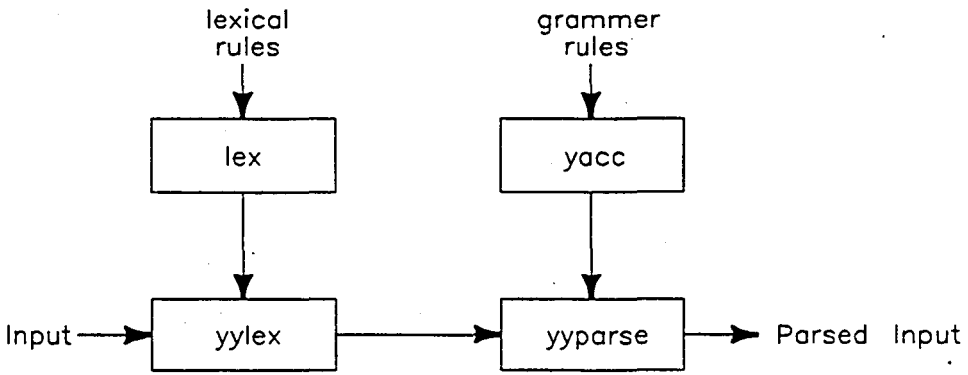


Figure 7-1. An Overview of `lex`

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%                          A space is required before \t.
[ \t]+$          ;
```

is all that is required. The program contains a **%%** delimiter to mark the beginning of the rules and one rule. This rule contains a regular expression which matches one or more instances of the characters *blank* or *tab* (written `\t` for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the **+** indicates "one or more ... "; and the **$** indicates "end of line," similar to `vi`. No action is specified, so the program generated by **lex** (**yylex**) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$             ;
[ \t]+ printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

lex: A Lexical Analyzer and Generator

The **lex** command can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. The **lex** command can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface **lex** and **yacc**. The **lex** programs recognize only regular expressions; **yacc** writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of **lex** and **yacc** is often appropriate. When used as a preprocessor for a later parser generator, **lex** is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 7-2. Additional programs, written by other generators or by hand, can be added easily to programs written by **lex**. The **yacc** command users will realize that the name **yylex** is what **yacc** expects its lexical analyzer to be named, so that the use of this name by **lex** simplifies interfacing.

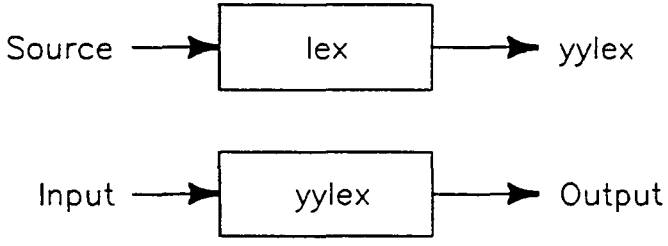


Figure 7-2. Using lex with yacc

The `lex` command generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a `lex` program to recognize and partition an input stream is proportional to the length of the input. The number of `lex` rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of re-scanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by `lex`.

In the program written by `lex`, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered, as cases of a switch statement in C. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

The `lex` command is not limited to source which can be interpreted on the basis of one character look-ahead. For example, if there are two rules, one looking for `ab` and another for `abcdefg`, and the input stream is `abcdefh`, `lex` will recognize `ab` and leave the input pointer just before `cd` ... Such backup is more costly than the processing of simpler languages.

lex Source

The general format of lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see the section "lex Regular Expression") and the right column contains *actions*, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer printf("found keyword INT");
```

to look for the string integer in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces.

As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. The `lex` command rules such as

```
colour      printf("color");
mechanise   printf("mechanize");
petrol      printf("gas");
```

would be a start. These rules are not quite enough, since the word `petroleum` would become `gaseum`. A way of dealing with this will be described later.

lex Regular Expressions

The definitions of regular expressions are similar to those in *ed(1)*. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string `integer` wherever it appears and the expression

```
a57D
```

looks for the string `a57D`.

Operators

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } %  < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string `xyz++` when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to `lex` lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the previous expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but space, tab, newline and the list above is always a text character.

Note that the initial percent operator (%) is special because it is the separator for `lex` source segments.

Character Classes

Classes of characters can be specified using the operator pair [ ]. The construction [abc] matches a single character, which may be a, b, or c. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \, -, and ^. The - character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underscore. Ranges may be given in either order. Using - between any pair of characters which are not both uppercase letters, both lowercase letters, or both digits is implementation dependent and will get a warning message. (For example, [0-z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character - in a character class, it should be first or last; thus

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

```
[^abc]
```

matches all characters except a, b, or c, including all special or control characters; or

```
[^a-zA-Z]
```

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary Character

To match almost any character, the operator character

```
(dot or period)
```

is the class of all characters except newline. Escaping into octal is possible although non-portable:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

7

Optional Expressions

The operator ? indicates an optional element of an expression. Thus

```
ab?c
```

matches either ac or abc.

Repeated Expressions

Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive a characters, including zero; while

```
a+
```

is one or more instances of a. For example,

```
[a-z]+
```

is all strings of lowercase letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping

The operator | indicates alternation:

```
(ab|cd)
```

matches either ab or cd. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab|cd+)?(ef)*
```

matches such strings as abefef, efefef, cdef, or cddd; but not abc, abcd, or abcdef.

Context sensitivity

The `lex` command will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [ ] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string ab, but only if followed by cd. Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in `lex` by `start conditions` as explained in the section on left context sensitivity. If a rule is only to be executed when the `lex` automaton interpreter is in start condition `x`, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition `ONE`, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Repetitions and Definitions

The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named `digit` and inserts it at that point in the expression. The definitions are given in the first part of the `lex` input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of a.

```
a{2, }
```

matches two or more occurrences of a, while

```
a{3}
```

matches exactly three occurrences of a and is equivalent to aaa.

Operator Precedence

The `lex` command operators are handled according to the following rules of precedence:

- The `lex` command operators are ranked in the following order of precedence, beginning with highest precedence and proceeding to the lowest precedence:
- All operations on a single line have the same precedence.
  - * ? +
  - concatenation
  - repetition
  - $ ^
  - |
  - / <>

lex Actions

When an expression is matched, **lex** executes the corresponding action. This section describes some features of **lex** which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus, the **lex** user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When **lex** is being used with **yacc**, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ";" as an action causes this result. A frequent rule is

```
[ \t\n] ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "         |
"\t"        |
"\n"        ;
```

with the same result, although in different style. The quotes around\n and\t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like **[a-z]+**. The **lex** command leaves this text in an external character array named **yytext**. Thus, to print the name found, a rule like

```
[a-z]+  printf("%s", yytext);
```

will print the string in **yytext**. The C function **printf** accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and **s** indicating string type), and the data is the

characters in yytext. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+  ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches read it will normally match the instances of read contained in bread or readjust; to avoid this, a rule of the form [a-z]+ is needed. This is explained below.

Sometimes it is more convenient to know the end of what has been found; hence lex also provides a count yyleng of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a-zA-Z]+            {words++; chars += yyleng;}
```

which accumulates in chars the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1]
```

Occasionally, a lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, yymore() can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in yytext. Second, yyless (*n*) may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in yytext to be retained. Further characters previously matched are returned to the input. This provides the same sort of look ahead offered by the / operator, but in a different form.

Example

Consider a language which defines a string as a set of characters between double quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression that matches such a string is somewhat confusing, so you might prefer to use:

```
\"[^"]*        {
        if (yytext[yyleng-1] == '\\')
             yymore();
        else
                ... normal user processing
        }
```

which will, when faced with a string such as `"abc\"def "`, first match the five characters `"abc\` ; then the call to `yymore()` will cause the next part of the string, `"def` , to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal user processing."

The function `yyless()` might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of `=-a` (In early versions of C, the assignment operators had the form `=op`. C now writes assignment operators in the form `op=` to avoid the ambiguity illustrated here.) Suppose it is desired to treat this as `=- a` but print a message. A rule might be

```
=-[a-zA-Z]      {
                printf("Operator (=-) ambiguous\n");
                yyless(yyleng-1);
                ... action for =- ...
                }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as `=-`. Alternatively it might be desired to treat this as `= -a`. To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]      {
                printf("Operator (=-) ambiguous\n");
                yyless(yyleng-2);
                ... action for = ...
                }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

7
```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity.

In addition to these routines, `lex` also permits access to the I/O routines it uses. They are:

1. `input()` which returns the next input character;
2. `output(c)` which writes the character `c` on the output; and
3. `unput(c)` pushes the character `c` back onto the input stream to be read later by `input()`.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by `input` must mean end of file; and the relationship between `unput` and `input` must be retained or the `lex` look-ahead will not work.

The `lex` command does not look ahead at all if it does not have to, but every rule ending in `+`, `\*`, `?`, or `$` or containing `/` implies look-ahead. Look-ahead is also necessary to match an expression that is a prefix of another expression. For a discussion of the character set used by `lex`, read the section "Character Set" found in this manual. The standard `lex` library imposes a 100 character limit on backup.

Another `lex` library routine that the user will sometimes want to redefine is `yywrap()` which is called whenever `lex` reaches an end-of-file. If `yywrap` returns a `1`, `lex` continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a `yywrap` which arranges for new input and returns `0`. This instructs `lex` to continue processing. The default `yywrap` always returns `1`.

This routine is also a convenient place to print tables, summaries, etc., at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through `yywrap`. In fact, unless a private version of `input()` is supplied a file containing nulls cannot be handled, since a value of 0 returned by `input` is taken to be end-of-file.

Ambiguous Source Rules

The `lex` command can handle ambiguous specifications. When more than one expression can match the current input, `lex` chooses as follows:

1. The longest match is preferred.
2. Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer keyword action ...;
[a-z]+  identifier action ...;
```

to be given in that order. If the input is `integers`, it is taken as an identifier, because `[a-z]+` matches 8 characters while `integer` matches only 7. If the input is `integer`, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. `int`) will not match the expression `integer` and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like `.*` dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

7

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, will stop after `'first'`. The consequences of errors like this are mitigated by the fact that the `.` operator will not match newline. Thus expressions like `.*` stop on the current line. Don't try to defeat this with expressions like `[.\n]+` or equivalents; the `lex` generated program will try to read the entire input file, causing internal buffer overflows.

Note that lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both she and he in an input text. Some lex rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides he and she. Remember that . does not include newline. Since she includes he, lex will normally *not* recognize the instances of he included in she, since once it has passed a she those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of he:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that she includes he but not vice versa, and omit the REJECT action on he; in other cases, however, it would not be possible *a priori* to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+  { ... ; REJECT;}
a[cd]+  { ... ; REJECT;}
```

If the input is ab, only the first rule matches, and on ad only the second matches. The input string accb matches the first rule for four characters and

then the second rule for three characters. In contrast, the input accd agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digraph table of the input is desired; normally the digraphs overlap, that is the word the is considered to contain both th and he. Assuming a two-dimensional array named digraph to be incremented, the appropriate source is

```
%%
    [a-z][a-z]    {digraph[yytext[0]][yytext[1]]++; REJECT;}
    .             ;
    \n            ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

lex Source Definitions

Remember the format of the lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by lex. These can go either in the definitions section or in the rules section.

Remember that lex is turning the rules into a program. Any source not intercepted by lex is copied into the generated program. There are three classes of such things.

1. Any line which is not part of a lex rule or action which begins with a blank or tab is copied into the lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by lex which contains the actions. This material must look like program fragments, and should precede the first lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the lex source or the generated code. The comments should follow the host language convention.

2. Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of formats, etc., is copied out after the lex output.

Definitions intended for `lex` are given before the first `%%` delimiter. Any line in this section not contained between `%{` and `%}`, and beginning in column 1, is assumed to define `lex` substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the *{name}* syntax in a rule. Using `{D}` for the digits and `{E}` for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                       [0-9]
E                       [DEde][-+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}                 printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as `35.EQ.I`, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ      printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within `lex` itself for larger source programs. These possibilities are discussed below under "Summary of Source Format."

Usage

There are two steps in compiling a `lex` source program. First, the `lex` source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of `lex` subroutines. The generated program is in a file named `lex.yy.c`. The I/O library is defined in terms of the C standard library.

HP-UX

The library is accessed by the loader flag `-ll` for C, so an appropriate set of commands is

```
lex source
cc lex.yy.c -ll
```

The resulting program is placed in the usual file `a.out` for later execution. To use `lex` with `yacc` see below. Although the default `lex` I/O routines use the C standard library, the `lex` automata themselves do not do so; if private versions of `input`, `output` and `unput` are given, the library can be avoided.

lex and yacc

If you want to use **lex** with **yacc**, note that what **lex** generates is a program named **yylex()**, the name required by **yacc** for its lexical analyzer. Normally, the default main program in the **lex** library calls this routine, but if **yacc** is loaded, and its main program is used, **yacc** will call **yylex()**. In this case each **lex** rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to **yacc**'s names for tokens is to compile the **lex** output file as part of the **yacc** output file by placing the line

```
# include "lex.yy.c"
```

in the last section of **yacc** input. If the grammar is **gram.y** and the lexical rules are **scan.l** the HP-UX command sequence can just be:

```
yacc gram.y
lex scan.l
cc y.tab.c -ly -ll
```

The **yacc** library (**ly**) should be loaded before the **lex** library, to obtain a main program which invokes the **yacc** parser. The generations of **lex** and **yacc** programs can be done in either order.

Alternatively, the **-d** option of **yacc** can be used to generate a file **y.tab.h** of token definitions. This can be included in the **lex** program by placing

```
%{
#include "y.tab.h"
%}
```

in the definitions section of the **lex** input file. If the grammar is **gram.y** and the lexical rules are in file **scan.l**, the HP-UX command sequence is:

```
yacc -d gram.y
lex scan.l
cc y.tab.c lex.yy.c -ly -ll
```

Examples

As a simple example, consider copying an input file while adding 3 to every positive number that is divisible by 7. Here is a suitable `lex` source program

```
%%
        int k;
[0-9]+  {
        k = atoi(yytext);
        if (k%7 == 0)
             printf("%d", k+3);
        else
             printf("%d",k);
        }
```

to do just that. The rule `[0-9]+` recognizes strings of digits; `atoi` converts the digits to binary and stores the result in `k`. The operator `%` (remainder) is used to check whether `k` is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as `49.63` or `X7`. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
        int k;
-?[0-9]+            {
                    k = atoi(yytext);
                    printf("%d", (k > 0 && k%7 == 0) ? k+3 : k);
                    }
-?[0-9.]+                   ECHO;
[A-Za-z][A-Za-z0-9]+        ECHO;
```

Numerical strings containing a `.` or preceded by a letter will be picked up by one of the last two rules, and not changed. The `if-else` has been replaced by a C conditional expression to save space; the form `a?b:c` means "if a then b else c."

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
     int lengs[100]; /*  Because this line has leading blanks,
                      *  it is copied to the lex.yy.c file (read
                      *  the section "lex Source Definitions."
                      */
%%
[a-z]+  lengs[yyleng]++;
.       |
\n      ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
     if (lengs[i] > 0)
          printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement `return(1);` indicates that `lex` is to perform wrapup. If `yywrap` returns zero (false) it implies
7 that further input is available and the program is to continue reading and processing. To provide a `yywrap` that never returns true causes an infinite loop.

As a larger example, *here are some program fragments which convert* double precision FORTRAN to single precision FORTRAN. Because FORTRAN does not distinguish uppercase and lowercase letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes `double precision` to `real`, or `DOUBLE PRECISION` to `REAL`.

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
  printf(yytext[0]=='d'? "real" : "REAL");
  }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^" "{5} [^ 0]    ECHO;
```

In the regular expression, the quotes surround the blank and are follow by the repeat operator {5}. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of ^. Here are some rules to change double precision constants to ordinary floating constants:

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+          |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+    |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+    {
     /* convert constants */
     for(p=yytext; *p != 0; p++)
          {
          if (*p == 'd')
              *p == 'e';
          else if (*p == 'D')
              *p = 'E';
          ECHO;
          }
```

After the floating point constant is recognized, it is scanned by the `for` loop to find the letter `d` or `D` converting it to `e` or `E`, respectively. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial `d`. By using the array `yytext` the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}      |
{d}{c}{o}{s}      |
{d}{s}{q}{r}{t}   |
{d}{a}{t}{a}{n}   |
...
{d}{f}{l}{o}{a}{t}      printf("%s",yytext+1);
```

Another list of names must have initial d changed to initial a:

```
{d}{l}{o}{g}      |
{d}{l}{o}{g}10    |
{d}{m}{i}{n}1     |
{d}{m}{a}{x}1     {
                  yytext[0] =+ 'a' - 'd';
                  ECHO;
                  }
```

And one routine must have initial d changed to initial r:

```
{d}1{m}{a}{c}{h}        {yytext[0] =+ 'r'  - 'd';
                        ECHO;
                        }
```

To avoid such names as **dsinx** being detected as instances of **dsin**, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
\n                      |
.                       ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

Left-Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since `lex` is not involved at all. It may be more convenient, however, to have `lex` remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when `lex` is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word `magic` to `first` on every line which began with the letter `a`, changing `magic` to `second` on every line which began with the letter `b`, and changing `magic` to `third` on every line which began with the letter `c`. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic   {
        switch (flag)
          {
            case 'a': printf("first"); break;
            case 'b': printf("second"); break;
            case 'c': printf("third"); break;
            default: ECHO; break;
          }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to `lex` in the definitions section with a line reading

```
%Start  name1 name2 ...
```

where the conditions may be named in any order. The word `Start` can be abbreviated to `s` or `S`. The conditions can be referenced at the head of a rule with the `<>` brackets:

```
<name1>expression
```

is a rule which is only recognized when `lex` is in the start condition *name1*. To enter a start condition, execute the action statement:

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state, use:

```
BEGIN 0;
```

or

```
BEGIN INITIAL
```

which resets the initial condition of the `lex` automaton interpreter.

A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active while in the normal state or any `%s` state. To specify that a rule is active *only* in the normal state, prefix it with `<INITIAL>`. Note that `INITIAL` is predefined by `lex`, and should not be included in a `%start` declaration.

The same example as before can be written:

```
%START AA BB CC
%%
^a      {ECHO; BEGIN AA;}
^b      {ECHO; BEGIN BB;}
^c      {ECHO; BEGIN CC;}
\n      {ECHO; BEGIN 0;}
<AA>magic       printf("first");
<BB>magic       printf("second");
<CC>magic       printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but `lex` does the work rather than the user's code.

The `lex` command also allows the definition of *exclusive start conditions*. The syntax is similar to that for *start conditions*, but the conditions are declared using `%x` or `%X`. For example:

```
%x name1, name2, ...
```

Exclusive start conditions differ from *start conditions* in how rules not beginning with the <> prefix operator are handled. When in a `%x` state, only rules explicitly prefixed by that *<state>* are active. Any rule not beginning with the <> prefix operator is not active.

The following example uses the **%x** exclusive start state. In this example, the scanner is looking for the keywords **first** and **second**; however, the symbol **##** puts the scanner into a "literal" mode where the normal patterns are not recognized. In this case, everything is just echoed (the default action) until another **##** symbol is reached that will put the scanner back into the initial state.

```
%x  LITERAL
%%
first  printf("FIRST");
second printf("SECOND");
## { BEGIN LITERAL; }
<LITERAL>##  { BEGIN INITIAL; }
```

Character Set

The programs generated by `lex` handle character I/O only through the routines `input`, `output`, and `unput`. Thus the character representation provided in these routines is accepted by `lex` and employed to return values in *yytext*. For internal use, a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter a is represented as the same form as the character constant `'a'`. If this interpretation is changed, by providing I/O routines which translate the characters, `lex` must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only `%T`. The table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. Thus the next example maps the lowercase and uppercase letters `a` through `z` together into the integers `1` through `26`, the newline character into `27`, `+` and `-` into `28` and `29` respectively, and the digits `0` through `9` into `30` through `39`. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character can be assigned the number 0, and no character can be assigned a number that exceeds the size of the hardware character set.

```
%T
 1        Aa
 2        Bb
...
26        Zz
27        \n
28        +
29        -
30        0
31        1
...
39        9
%T
```

Summary of Source Format

The general form of a `lex` source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1. Definitions, in the form "name space translation".
2. Included code, in the form "space code".
3. Included code, in the form

```
%{
code
%}
```

4. Start conditions, given in the form

```
%S name1 name2 ...
```

5. Character set tables, in the form

```
%T
number space character-string
...
%T
```

6. Changes to internal array sizes, in the form

```
%x  nnn
```

where nnn is a decimal integer representing an array size and **x** selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in `lex` use the following operators:

x	the character "*x*"
"*x*"	an "*x*", even if *x* is an operator.
x	an "*x*", even if *x* is an operator.
[*xy*]	the character *x* or *y*.
[*x*-*z*]	the characters *x*, *y* or *z*.
[^*x*]	any character but *x*.
.	any character but newline.
^*x*	an *x* at the beginning of a line.
<*y*>*x*	an *x* when `lex` is in start condition *y*.
x$	an *x* at the end of a line.
x?	an optional *x*.
*x**	0,1,2, ... instances of *x*.
x+	1,2,3, ... instances of *x*.
x|*y*	an *x* or a *y*.
(*x*)	an *x*.
x/*y*	an *x* but only if followed by *y*.
{*xx*}	the translation of *xx* from the definitions section.
x{*m*,*n*}	*m* through *n* occurrences of *x*

Caveats and Known Defects

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used unput to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

8

yacc: Yet Another Compiler-Compiler

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an *input language* which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

The yacc command provides a general tool for describing the input to a computer program. The yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. The yacc command turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

The yacc command is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, Ratfor, etc., yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a FORTRAN debugging system.

Introduction

The yacc command provides a general tool for imposing structure on the input to a computer program. The yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. The yacc command then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

The yacc command is written in a portable dialect of C and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year;
```

Here, date, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and year are defined elsewhere. The comma , is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user supplied routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name  :  'J' 'a' 'n'   ;
month_name  :  'F' 'e' 'b' ;

. . .

month_name  :  'D' 'e' 'c' ;
```

might be used in the previous example. The lexical analyzer would only need to recognize individual letters, and `month_name` would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a `month_name` was seen; in this case, `month_name` would be a token.

Literal characters such as , must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date  :  month '/' day '/' year   ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying yacc has been described elsewhere [2] [3] [4]. The yacc command has been extensively used in numerous practical applications, including `lint` [5], the Portable C Compiler [6], and a system for typesetting mathematics [7].

The next several sections describe the basic process of preparing a yacc specification.

Sections in this Chapter	Description
"Basic Specifications"	Explains the preparation of grammar rules
"Actions"	Covers the preparation of the user supplied actions associated with the grammar rules
"Lexical Analysis"	Explains the preparation of lexical analyzers
"How the Parser Works"	Covers the operation of the parser
"Ambiguity and Conflicts"	Gives reasons why yacc may be unable to produce a parser from a specification, and what to do about it
"Precedence and Associativity"	Provides a simple mechanism for handling operator precedences in arithmetic expressions

Sections in this Chapter	Description
"Error Handling"	Covers error detection and recovery
"The yacc Environment"	Covers the operating environment and special features of the parsers yacc produces
"Hints for Debugging"	Explains how to debug a yacc grammar
"Hints for Preparing Specifications"	Gives some suggestions which should improve the style and efficiency of the specifications
"Advanced Topics"	Covers advanced features of yacc
"yacc Examples, Input Syntax, and Support"	Provides yacc examples, input syntax, and support information
"Acknowledgements"	Gives credit to those people who contributed to yacc

Basic Specifications

Names refer to either tokens or nonterminal symbols. The yacc command requires token names to be declared as such. In addition, for reasons discussed in the section "Lexical Analysis," it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent %% marks. (The percent % is generally used in yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* ... */, as in C.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

8

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot (.), underscore (_), and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes. As in C, the backslash (\) is an escape character within literals, and all the C escapes are recognized. Thus

```
'\n'        newline
'\r'        return
'\''        single quote ''''
'\\'        backslash ''\''
'\t'        tab
'\b'        backspace
'\f'        form feed
'\xxx'      ''xxx'' in octal
```

For a number of technical reasons, the NULL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, the vertical bar (|) can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A       :       B  C  D   ;
A       :       E  F   ;
A       :       G   ;
```

can be given to yacc as

```
A       :       B  C  D
        |       E  F
        |       G
;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :    ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token   name1  name2 . . .
```

in the declarations section. For more information, see the sections "Lexical Analysis", "Ambiguity and Conflicts", and "Precedence." Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the `%start` keyword:

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see the section "Lexical Analysis," below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces { and }. For example:

```
A       :       '('  B  ')'   {       hello( 1, "abc" );  }
```

and

```
XXX     :       YYY  ZZZ
                        {       printf("a message\n");
                                flag = 25;   }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" **$** is used as a signal to yacc in this context.

To return a value, the action normally sets the pseudo-variable **$$** to some value. For example, an action that does nothing but return the value 1 is

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, ... , which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A       :       B  C  D   ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr    :       '('  expr  ')'   ;
```

The value returned by this rule is usually the value of the **expr** in parentheses. This can be indicated by

```
expr    :       '('  expr  ')'              {  $$ = $2 ;  }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A       :       B    ;
```

frequently need not have an explicit action. This last rule is equivalent to

```
A:      B
          { $$ = $1; }
```

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. The yacc command permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A       :       B
                        {  $$ = 1;  }
                C
              {   x = $2;   y = $3;  }
        ;
```

the effect is to set **x** to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually handled by yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. The yacc command actually treats the above example as if it had been written:

```
$ACT    :       /* empty */
                        {  $$ = 1;  }
        ;

A       :       B  $ACT  C
                        {   x = $2;   y = $3;  }
        ;
```

A good understanding of how yacc handles actions can be important when interpreting conflict messages for such rules (see the section "Ambiguity and Conflicts"). For example, conflicts in the grammar specification occur when an interior action occurs in a rule before the parser can be sure which rule is being reduced.

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function node, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. The parse tree can be built by supplying actions such as:

```
expr    :       expr  '+'  expr
                          { $$ = node('+', $1, $3); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks %{ and %}. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   int variable = 0;   %}
```

could be placed in the declarations section, making variable accessible to all of the actions. The yacc parser uses only names beginning in yy for parser variables; the user should avoid such names.

In these examples, all the values returned by actions and values of tokens are integers; a discussion of values of other types will be found in the section "Advanced Topics."

8

Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called **yylex**. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable **yylval**.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by yacc, or chosen by the user. In either case, the **# define** mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                    . . .
        case '0':
        case '1':
          . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                . . .
                }
        . . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names `if` or `while` will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name `error` is reserved for error handling, and should not be used naively (see the section "Error Handling").

As mentioned above, the token numbers may be chosen by `yacc` or by the user. In the default situation, the numbers are chosen by `yacc`. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

The `lex` program is a very useful tool for constructing lexical analyzers. Lexical analyzers are designed to work in close harmony with `yacc` parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. `lex` can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

How the Parser Works

The yacc command turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called `shift`, `reduce`, `accept`, and `error`. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls `yylex` to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The `shift` action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF          shift 34
```

which says, in state 56, if the lookahead token is `IF`, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The `reduce` action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the

rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
. . reduce 18
```

refers to grammar rule 18, while the action

```
IF          shift 34
```

refers to state 34.

Suppose the rule being reduced is

```
A       : x y z    ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is, in effect, a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a goto action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A          goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable `yylval` is copied onto the value stack. After the return from the user code, the reduction is carried out. When the `goto` action is done, the external variable `yyval` is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The `accept` action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The `error` action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in the section "Error Handling."

It is time for an example! Consider the specification

```
%token  DING  DONG  DELL
%%
rhyme   :       sound  place
        ;
sound   :       DING  DONG
        ;
place   :       DELL
        ;
```

When yacc is invoked with the -v option, a file called y.output is produced, with a human-readable description of the parser. The y.output file corresponding to the previous grammar (with some statistics stripped off the end) is:

```
state 0
        $accept  :  _rhyme  $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept  :   rhyme_$end

        $end  accept
        .  error

state 2
        rhyme  :   sound_place

        DELL  shift 5
        .  error

        place   goto 4

state 3
        sound   :   DING_DONG

        DONG  shift 6
        .  error
```

```
state 4
        rhyme  :   sound  place_    (1)

        .   reduce  1

state 5
        place  :   DELL_    (3)

        .   reduce  3

state 6
        sound  :   DING  DONG_    (2)

        .   reduce  2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING  DONG  DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is **state 0**. The parser needs to refer to the input in order to decide between the actions available in **state 0**, so the first token, DING, is read, becoming the lookahead token. The action in **state 0** on DING is **shift 3**, so **state 3** is pushed onto the stack, and the lookahead token is cleared. Then **state 3** becomes the current state. The next token, DONG, is read, becoming the lookahead token. The action in **state 3** on the token DONG is **shift 6**, so **state 6** is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In **state 6**, without even consulting the lookahead, the parser reduces by rule 2.

```
sound  :  DING  DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering **state 0**. Consulting the description of **state 0**, looking for a **goto** on **sound**,

```
sound   goto 2
```

is obtained; thus **state 2** is pushed onto the stack, becoming the current state.

In **state 2**, the next token, DELL, must be read. The action is **shift 5**, so **state 5** is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In **state 5**, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one **state**, 5, is popped off, and **state 2** is uncovered. The goto in **state 2** on *place*, the left side of rule 3, is **state 4**. Now, the stack contains 0, 2, and 4. In **state 4**, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering **state 0** again. In **state 0**, there is a goto on **rhyme** causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by **$end** in the **y.output** file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, etc. A few minutes spent with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr        :     expr  '-'  expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr  -  expr  -  expr
```

the rule allows this input to be structured as either

```
(  expr  -  expr  )  -  expr
```

or as

```
expr  -  (  expr  -  expr  )
```

(The first is called *left association*, the second *right association*).

The yacc command detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr  -  expr  -  expr
```

When the parser has read the second **expr**, the input that it has seen:

```
expr  -  expr
```

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule; after applying the rule; the input is reduced to **expr** (the left side of the rule). The parser would then read the final part of the input:

```
-  expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr  -  expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to `expr` and leaving

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a **`shift / reduce`** conflict. It may also happen that the parser has a choice of two legal reductions; this is called a **`reduce / reduce`** conflict. Note that there are never any **`shift/shift`** conflicts.

When there are **`shift/reduce`** or **`reduce/reduce conflicts`**, `yacc` still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule*.

The `yacc` command invokes two disambiguating rules by default:

- In a **`shift/reduce`** conflict, the default is to do the shift.
- In a **`reduce/reduce`** conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but **`reduce/reduce`** conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than `yacc` can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads

to an incorrect parser. For this reason, yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by rule 1 and rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an if-then-else construction:

```
stat    :       IF  '('  cond  ')'  stat_
        |       IF  '('  cond  ')'  stat_ELSE  stat
        ;
```

In these rules, IF and ELSE are tokens, cond is a nonterminal symbol describing conditional (logical) expressions, and stat is a nonterminal symbol describing statements. The first rule will be called the simple-if rule, and the second the if-else rule.

These two rules form an ambiguous construction, since input of the form

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be structured according to these rules in two ways:

```
IF  (  C1  )  {
              IF  (  C2  )  S1
              }
ELSE  S2
```

or

```
IF  (  C1  )  {
              IF  (  C2  )  S1
              ELSE  S2
              }
```

The second interpretation is the one given in most programming languages having this construct. Each ELSE is associated with the last preceding IF that yet doesn't have a corresponding ELSE. In this example, consider the situation where the parser has seen

```
IF  (  C1  )  IF  (  C2  )  S1
```

and is looking at the ELSE. It can immediately reduce by the simple-if rule to get

```
IF  (  C1  )  stat
```

and then read the remaining input,

```
ELSE  S2
```

and reduce

```
IF  (  C1  )  stat  ELSE  S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted, S2 read, and then the right hand portion of

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be reduced by the if-else rule to get

```
IF  (  C1  )  stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – there is a `shift/reduce` conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This `shift/reduce` conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

```
IF  (  C1  )  IF  (  C2  )  S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of yacc are best understood by examining the verbose (-v) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

        stat  :  IF  (  cond  )  stat_           (18)
        stat  :  IF  (  cond  )  stat_  ELSE  stat

        ELSE     shift 45
        .        reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF  (  cond  )  stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is ELSE, it is possible to shift into state 45. Part of the description for state 45 is the line:

```
stat  :  IF  (  cond  )  stat  ELSE_stat
```

since the ELSE will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not ELSE, the parser reduces by grammar rule 18:

```
stat  :  IF  '('  cond  ')'  stat
```

8

Once again, notice that the numbers following **shift** commands refer to other states, while the numbers following **reduce** commands refer to grammar rule numbers. In the **y.output** file, the rule numbers are printed after those rules which can be reduced. In most states, there will be at most one **reduce** action possible in the state, and this will be the default command. The user who encounters unexpected **shift/reduce** conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references[234] might be consulted; the services of a local guru might also be appropriate.

Precedence and Associativity

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a yacc keyword: `%left`, `%right`, or `%nonassoc`, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

8 describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than asterisk (*) and slash (/), which are also left associative. The keyword `%right` is used to describe right associative operators, and the keyword `%nonassoc` is used to describe operators, like the operator `.LT.` in FORTRAN, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in FORTRAN, and such an operator would be described with the keyword %nonassoc in yacc. As an example of the behavior of these declarations, the description

```
%right  '='
%left   '+'  '-'
%left   '*'  '/'

%%

expr    :       expr  '='  expr
        |       expr  '+'  expr
        |       expr  '-'  expr
        |       expr  '*'  expr
        |       expr  '/'  expr
        |       NAME
        ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '–'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left  '+'  '-'
%left  '*'  '/'

%%

expr    :       expr  '+'  expr
        |       expr  '-'  expr
        |       expr  '*'  expr
        |       expr  '/'  expr
        |       '-'  expr      %prec  '*'
        |       NAME
        ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of `shift/reduce` and `reduce/reduce` conflicts reported by `yacc`. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The `y.output` file is very useful in deciding whether the parser is actually doing what was intended.

Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, yacc provides a simple, but reasonably general, feature. The token name `error` is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token `error` is legal. It then behaves as if the token `error` were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat    :    error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :       error   ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input : error '\n' { printf( "Reenter last line: " ); } input
                   { $$  =  $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input :  error  '\n'
                {       yyerrok;
                        printf( "Reenter last line: " );   }
         input
                {       $$  =  $4;  }
       ;
```

As mentioned above, the token seen immediately after the error symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by **yylex** would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :       error
                        {       resynch();
                                yyerrok ;
                                yyclearin ;     }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

The yacc Environment

When the user inputs a specification to yacc, the output is a file of C programs, called `y.tab.c`. The function produced by yacc is called `yyparse`. It is an integer valued function. When it is called, it in turn repeatedly calls `yylex`, the lexical analyzer supplied by the user (see Chapter 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) `yyparse` returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, `yyparse` returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called `main` must be defined, that eventually calls `yyparse`. In addition, a routine called `yyerror` prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using yacc, a library has been provided with default versions of `main` and `yyerror`. The name of this library is system dependent; on HP-UX systems the library is accessed by a `-ly` argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
      return( yyparse() );
      }
```

and

```
# include stdio.h

yyerror(s) char *s; {
                    fprintf( stderr, "%s\n", s );
                    }
```

The argument to yyerror is a string containing an error message, usually the string syntax error. The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable yychar contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the main program is probably supplied by the user (to read arguments, etc.) the yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable yydebug is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

The yacc command provides command line options to allow some user modifications to the yacc environment. The -b option allows the user to change the default output filenames. For example:

```
yacc -b xx gram.y
```

would produce a file xx.tab.c rather than y.tab.c.

The -p option specifies a prefix to replace yy in naming external procedures and variables. This can aid the linking of multiple parsers into a single program. Note that if this renaming did not take place, multiple declaration errors would occur at link time. The names changed are yyparse, yylex, yyerror, yylval, yychar, and yydebug. The renaming is done by emitting a set of #defines in the y.tab.c and y.tab.h files.

Hints for Debugging

Debugging a yacc grammar can be a challenge since the user's code (the yacc grammar specification) is a level of abstraction away from the C code that is being debugged. There are generally two areas where run time debugging is needed:

- determining that the grammar is parsing its input as expected
- debugging action code.

To aid in debugging the parsing, yacc provides a trace facility. The trace is enabled by compiling `y.tab.c` with the preprocessor symbol `YYDEBUG` defined to be nonzero. This is easily done by using the `-t` option of yacc:

```
yacc -t grammar source
```

Then the trace is turned on by setting the variable **yydebug** to a nonzero value. This can be done either by editing `y.tab.c` before compiling it, or while debugging in **xdb** or **cdb**. The trace gives information about which parser shift and reduce actions are performed.

The action code and the overall program can be debugged using the symbolic debugger **xdb** or **cdb**, but some special work must be done to remove `# line` constructs that confuse the debuggers. Normally, yacc inserts `# line` constructs into the generated `y.tab.c` file so that compile time errors refer to lines in the yacc source file rather than the `y.tab.c` file. However, this causes problems for the symbolic debuggers **xdb** and **cdb** because the line numbers may not be in increasing order. Once the `y.tab.c` file has been successfully compiled, then run yacc with the `-l` option so that no `# line` constructs are generated. This `y.tab.c` file can then be debugged using either **xdb** or **cdb**. Remember that the code lines in either **xdb** or **cdb** refer to lines in the `y.tab.c` file, not the original yacc source file.

Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

1. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."
2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.
5. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The examples in the section "yacc Examples, Input Syntax, and Support" are written following this style, as are the examples in the text of this chapter (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name    :       name  rest_of_rule  ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |       list  ','  item
        ;
```

and

```
seq     :       item
        |       seq  item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq     :       item
        |       item  seq
        ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable. It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :       /* empty */ |  seq  item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
  ...  other declarations ...

%%

prog    :       decls  stats
        ;

decls   :       /* empty */
                        {       dflag = 1;  }
        |       decls  declaration
        ;

stats   :       /* empty */
                        {       dflag = 0;  }
        |       stats  statement
        ;

     ...  other rules ...
```

The flag `dflag` is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement*. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of backdoor approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like `if`, which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of '`if`' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

Advanced Topics

This section discusses a number of advanced features of yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes yyparse to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; yyerror is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider:

```
sent    :       adj  noun  verb  adj  noun
                {
look at the sentence
  . . .  }
        ;

adj     :       THE     {       $$ = THE;  }
        |       YOUNG   {       $$ = YOUNG;  }
        . . .
        ;

noun    :       DOG
                        {       $$ = DOG;  }
        |       CRONE
                        {       if( $0 == YOUNG ){
                                        printf( "what?\n" );
                                                }
                                $$ = CRONE;
                        }
        ;
        . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol noun in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. The yacc command can also support values of other types, including structures. In addition, yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The yacc value stack (see the section "How the Parser Works") is declared to be a union of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as `lint`[5] will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union  {
        body of union ...
        }
```

This declares the yacc value stack, and the external variables `yylval` and `yyval`, to have type equal to this union. If yacc was invoked with the `-d` option, the union declaration is copied onto the `y.tab.h` file. Alternatively, the union may be declared in a header file, and a `typedef` used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left   optype   '+'   '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type   nodetype   expr   stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no a priori type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :       aaa  {  $<intval>$  =  3;  } bbb
                     {       fun( $<intval>2, $<other>0 );  }
        ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in the section "An Advanced Example." The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or to refer to something with no defined type is diagnosed. If these facilities are not triggered, the yacc value stack is used to hold int's, as was true historically.

For parser efficiency, when an arbitrary **union** is defined for **YYSTYPE**, all of the members of the **union** should be kept to the size of an integer (for example, an integer or pointer). This is because the **yacc** value stack will be an array of these **union** types. If some **union** members are large, the stack will be large and copying stack values will be inefficient.

When larger structures are needed (for example, tree nodes in a compiler), it is recommended that allocation and deallocation of structures be handled by user supplied routines, and that the **union** member **YYSTYPE** be a pointer to the appropriate structure type.

yacc Examples, Input Syntax, and Support

This section contains the following information:

- An example of the yacc specification for a small desk calculator.
- An example of a grammar using some of yacc's advanced features.
- A description of the yacc input syntax.
- Old yacc features that are supported but not encouraged.

A Simple Example

This example gives the complete yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled a through z, and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules. This job is probably better done by the lexical analyzer.

```
%{
#  include  <stdio.h>
#  include  <ctype.h>

int  regs[26];
int  base;

%}

%start  list
```

```
%token  DIGIT  LETTER

%left  '|'
%left  '&'
%left  '+'  '-'
%left  '*'  '/'  '%'
%left  UMINUS      /*  supplies  precedence  for  unary  minus  */

%%                 /*  beginning  of  rules  section  */

list    :       /*  empty  */
        |       list  stat  '\n'
        |       list  error  '\n'
                        {       yyerrok;  }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 );  }
        |       LETTER  '='  expr
                        {       regs[$1]  =  $3;  }
        ;

expr    :       '('  expr  ')'
                        {       $$  =  $2;  }
        |       expr  '+'  expr
                        {       $$  =  $1  +  $3;  }
        |       expr  '-'  expr
                        {       $$  =  $1  -  $3;  }
        |       expr  '*'  expr
                        {       $$  =  $1  *  $3;  }
        |       expr  '/'  expr
                        {       $$  =  $1  /  $3;  }
        |       expr  '%'  expr
                        {       $$  =  $1  %  $3;  }
        |       expr  '&'  expr
                        {       $$  =  $1  &  $3;  }
        |       expr  '|'  expr
                        {       $$  =  $1  |  $3;  }
        |       '-'  expr       %prec  UMINUS
                        {       $$  =  -  $2;  }
        |       LETTER
```

```
                {       $$ = regs[$1]; }
        |       number
        ;

number  :       DIGIT
                {       $$ = $1; base = ($1==0) ? 8 : 10; }
        |       number DIGIT
                {       $$ = base * $1 + $2; }
        ;

%%              /* start of programs */

yylex() {               /* lexical analysis routine */
        /* returns LETTER for a lower case letter, yylval = 0
through 25 */
        /* return DIGIT for a digit, yylval = 0 through 9 */
        /* all other characters are returned immediately */

        int c;

        while( (c=getchar()) == ' ' ) { /* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                        yylval = c - 'a';
                        return ( LETTER );
                        }
        if( isdigit( c ) ) {
                        yylval = c - '0';
                        return( DIGIT );
                        }
        return( c );
        }
```

Advanced Example

This is an example of a grammar using some of the advanced features discussed in the section "Advanced Topics." The desk calculator example in the section "A Simple Example" in this section is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, a through z. Moreover, it also understands *intervals*, written

```
( x , y )
```

where x is less than or equal to y. There are 26 interval valued variables A through Z that may also be used. The usage is similar to that in the section "A Simple Example" in this appendix; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as double's. This structure is given a type name, INTERVAL, by using typedef. The yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through yacc: 18 shift/reduce and 26 reduce/reduce. The problem can be seen by looking at the two input lines:

```
2.5 + ( 3.5 - 4. )
```

and

```
2.5 + ( 3.5 , 4. )
```

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the , is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Also, while this example illustrates the use of arbitrary yacc stack value types, the union member `INTERVAL` results in every element of the yacc value stack being the size of a `struct` interval. For large structures, this can be very inefficient, and pointers to structures should be used instead. The user code must then supply routines to explicitly allocate and deallocate the structures.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine **atof** is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
                        double lo, hi;
                        } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start    lines

%union    {
          int ival;
          double dval;
          INTERVAL vval;
          }

%token  <ival>  DREG  VREG      /* indices into dreg, vreg arrays */

%token  <dval>  CONST           /* floating point constant */

%type   <dval>  dexp            /* expression */

%type   <vval>  vexp            /* interval expression */
```

```
        /*  precedence  information  about  the  operators  */

%left   '+'  '-'
%left   '*'  '/'
%left   UMINUS          /*  precedence  for  unary  minus  */

%%

lines   :       /*  empty  */
        |       lines  line
        ;

line    :       dexp  '\n'
                        {  printf( "%15.8f\n", $1 );}
        |       vexp  '\n'
                        { printf("(%15.8f,%15.8f)\n",$1.lo,$1.hi);}
        |       DREG  '='  dexp  '\n'
                        {     dreg[$1]  =  $3;}
        |       VREG  '='  vexp  '\n'
                        {     vreg[$1]  =  $3;}
        |       error  '\n'
                        {  yyerrok;}
        ;

dexp    :       CONST
        |       DREG
                        {       $$  =  dreg[$1];  }
        |       dexp  '+'  dexp
                        {       $$  =  $1  +  $3;  }
        |       dexp  '-'  dexp
                        {       $$  =  $1  -  $3;  }
        |       dexp  '*'  dexp
                        {       $$  =  $1  *  $3;  }
        |       dexp  '/'  dexp
                        {       $$  =  $1  /  $3;  }
        |       '-'  dexp       %prec  UMINUS
                        {       $$  =  - $2;  }
        |       '('  dexp  ')'
                        {       $$  =  $2;  }
        ;
```

```
vexp    :       dexp
                        {       $$.hi  =  $$.lo  =  $1;  }
        |       '('  dexp  ','  dexp  ')'
                        {
                        $$.lo  =  $2;
                        $$.hi  =  $4;
                        if($$.lo > $$.hi){
                                printf("interval out of order\n");
                                YYERROR;
                                }
                        }
        |       VREG
                        {       $$  =  vreg[$1];    }
        |       vexp  '+'  vexp
                        {       $$.hi  =  $1.hi  +  $3.hi;
                                $$.lo  =  $1.lo  +  $3.lo;}
        |       dexp  '+'  vexp
                        {       $$.hi  =  $1  +  $3.hi;
                                $$.lo  =  $1  +  $3.lo;}
        |       vexp  '-'  vexp
                        {       $$.hi  =  $1.hi  -  $3.lo;
                                $$.lo  =  $1.lo  -  $3.hi;}
        |       dexp  '-'  vexp
                        {       $$.hi  =  $1  -  $3.lo;
                                $$.lo  =  $1  -  $3.hi;}
        |       vexp  '*'  vexp
                        {       $$ = vmul( $1.lo, $1.hi, $3 );}
        |       dexp  '*'  vexp
                        {       $$  =  vmul(  $1,  $1,  $3  );}
        |       vexp  '/'  vexp
                        {       if(  dcheck(  $3  )  )  YYERROR;
                                $$  =  vdiv(  $1.lo,  $1.hi,  $3  );}
        |       dexp  '/'  vexp
                        {       if(  dcheck(  $3  )  )  YYERROR;
                                $$  =  vdiv(  $1,  $1,  $3  );}
        |       '-'  vexp       %prec  UMINUS
                        {       $$.hi  =  -$2.lo;    $$.lo  =  -$2.hi;}
        |       '('  vexp  ')'
                        {       $$  =  $2;}
        ;
```

```
%%

# define BSZ 50     /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register c;

        while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

        if( isupper( c ) ){
                        yylval.ival = c - 'A';
                        return( VREG );
                        }
        if( islower( c ) ){
                        yylval.ival = c - 'a';
                        return( DREG );
                        }

        if( isdigit( c ) || c=='.' ){
                /* gobble up digits, points, exponents */

                char buf[BSZ+1], *cp = buf;
                int dot = 0, exp = 0;

            for( ; (cp-buf)&<BSZ ; ++cp,c=getchar() ){

                        *cp = c;
                        if( isdigit( c ) ) continue;
                        if( c == '.' ){
                                        if(dot++ || exp) return( '.' );
                                        /* will cause syntax error */
                                          continue;
                                        }

                        if( c == 'e' ){
                                         if( exp++ ) return('e');
                                        /* will cause syntax error */
                                          continue;
```

```
                                        }

                        /*  end  of  number  */
                        break;
                        }
                *cp  =  '\0';
                if((cp-buf) >= BSZ) printf("constant too long: truncated\n");
                else ungetc( c, stdin );  /* push back last char read */
                yylval.dval  =  atof( buf );
                return(  CONST  );
                }
        return(  c  );
        }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and d */
        /*  used  by  *,  /  routines  */
        INTERVAL  v;

        if(  a>b  )  {  v.hi  =  a;    v.lo  =  b;  }
        else  {  v.hi  =  b;    v.lo  =  a;  }

        if(  c>d  )  {
                     if(  c>v.hi  )  v.hi  =  c;
                     if(  d<v.lo  )  v.lo  =  d;
                     }
        else         {
                     if(  d>v.hi  )  v.hi  =  d;
                     if(  c<v.lo  )  v.lo  =  c;
                     }
        return(  v  );
        }

INTERVAL  vmul(  a,  b,  v  )  double  a,  b;    INTERVAL  v;  {
        return(  hilo(  a*v.hi,  a*v.lo,  b*v.hi,  b*v.lo  )  );
        }

dcheck(  v  )  INTERVAL  v;  {
        if(  v.hi  >=  0.  &&  v.lo  <=  0.  ){
                     printf(  "divisor  interval  contains  0.\n"  );
                     return(  1  );
```

```
                }
        return(  0  );
        }

INTERVAL  vdiv(  a,  b,  v  )  double  a,  b;    INTERVAL  v;  {
        return(  hilo(  a/v.hi,  a/v.lo,  b/v.hi,  b/v.lo  )  );
        }
```

Input Syntax

This is a description of the yacc input syntax, as a yacc specification. Context dependencies, etc., are not considered. Ironically, the yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
            /*  grammar  for  the  input  to  ''yacc''  */

        /*  basic  entities  */
%token  IDENTIFIER      /* includes identifiers and literals */
%token  C_IDENTIFIER    /* identifier (but not literal) followed by
colon    */
%token  NUMBER          /*    [0-9]+    */

        /* reserved words: %type => TYPE, %left => LEFT, etc. */

%token  LEFT  RIGHT  NONASSOC  TOKEN  PREC  TYPE  START  UNION

%token  MARK    /*  the  %%  mark  */
%token  LCURL   /*  the  %{  mark  */
%token  RCURL   /*  the  %}  mark  */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs  MARK  rules  tail
        ;

tail    :       MARK { In this action, eat up the rest of the file }
```

```
        |       /*  empty:  the  second  MARK  is  optional  */
        ;

defs    :       /*  empty  */
        |       defs  def
        ;

def     :       START  IDENTIFIER
        |       UNION  {  Copy union  definition  to  output  }
        |       LCURL  {  Copy  C  code  to  output  file  }  RCURL
        |       ndefs  rword  tag  nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
        |       NONASSOC
        |       TYPE
        ;

tag     :       /*  empty:  union  tag  is  optional  */
        |       '<'  IDENTIFIER  '>'
        ;

nlist   :       nmno
        |       nlist  nmno
        |       nlist  ','  nmno
        ;

nmno    :       IDENTIFIER          /* NOTE: literal illegal with
                                         %type  */
        |       IDENTIFIER  NUMBER  /* NOTE: illegal  with %type */
        ;

        /*  rules  section  */

rules   :       C_IDENTIFIER  rbody  prec
        |       rules  rule
        ;

rule    :       C_IDENTIFIER  rbody  prec
```

```
        |       '|'  rbody  prec
        ;

rbody   :       /*  empty  */
        |       rbody  IDENTIFIER
        |       rbody  act
        ;

act     :       '{'  {  Copy  action,  translate  $$,  etc.<<  }  '}'
        ;

prec    :       /*  empty  */
        |       PREC  IDENTIFIER
        |       PREC  IDENTIFIER  act
        |       prec  ';'
        ;
```

8

Old Features Supported but Not Encouraged

This section mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

- Literals can also be delimited by double quotes ".
- Literals can be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

  The use of multi-character literals is likely to mislead those unfamiliar with yacc, since it suggests that yacc is doing a job which must be actually done by the lexical analyzer.
- Most places where % is legal, backslash \ may be used. In particular, \\ is the same as %%, `\left` the same as `%left`, etc.
- There are a number of other synonyms:
  - %< is the same as `%left`
  - %> is the same as `%right`
  - `%binary` and %2 are the same as `%nonassoc`
  - %0 and `%term` are the same as `%token`
  - %= is the same as `%prec`
- Actions can also have the form

  ={ ... }

  and the curly braces can be dropped if the action is a single C statement.
- C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Acknowledgements

B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder are to be recognized for their contribution of ideas to the current version of yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this chapter into English. Al Aho also deserves special credit for his help and favors.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys 6*(2) pp. 99-124 (June 1974).
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach. 18*(8) pp. 441-452 (August 1975).
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).
5. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (December 1977).
6. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).
7. B. W. Kernighan and L. L. Charry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach. 18* pp. 151-157 (March 1975).
8. M. E. Lesk, "Lex – A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975). (See *HP-UX Concepts and Tutorials*, Vol. 1.)

Index

D

M

Reorder No. or
Manual Part No.
92501-90005
E0892

Printed on
Recycled Paper

Printed in USA

92501-90005