

HP C/HP-UX Reference Manual

HP 9000 Computers

Ninth Edition



B3901-90003

HP-UX/HP C Compiler

June 2000

Legal Notice

Hewlett-Packard makes no warranty of any kind with regard to this manual, 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 direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Copyright © 2000 Hewlett-Packard Company.

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

Use, duplication or disclosure by the U.S. Government Department of Defense is subject to restrictions as set forth in paragraph (b)(3)(ii) of the Rights in Technical Data and Software clause in FAR 52.227-7013.

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

Use of this manual and flexible disc(s), compact disc(s), or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

UNIX is a registered trademark of The Open Group.

Contents

1. What is HP C?

ANSI Mode	2
Compatibility Mode	3
About HP C/HP-UX	4
HP C Online Help	5
Prerequisites for using HP C Online Help	5
Accessing HP C Online Help	6
X-Motif CDE Help is obsolete	6

2. Program Organization

Lexical Elements	8
White Space, Newlines, and Continuation Lines	8
Spreading Source Code Across Multiple Lines	8
Comments	9
Identifiers	9
Keywords	11
Declarations	19
Typedef Declarations	19
Abstract Global Types	20
Improving Portability	20
Simplifying Complex Declarations	21
Using typedefs for Arrays	21
Name Spaces	22
Declarations within code	23
Constants	24
Integer Constants	24
Floating-Point Constants	25
Character Constants	27
String Constants	29

Contents

Structuring a C Program	31
3. Data Types and Declarations	
Program Structure	34
Syntax	34
Description	34
Declarations	35
Syntax	35
Description	35
New Declaration features	36
Examples	37
Storage-Class Specifiers	38
Syntax	38
Description	38
Type Specifiers	40
Syntax	40
Description	40
HP-Specific Type Qualifiers	42
Syntax	42
Description	42
Type Qualifiers	44
Syntax	44
Description	44
Structure and Union Specifiers	47
Syntax	47
Description	47
Structure and Union Tags	49
Enumeration	52
Syntax	52

Contents

Description52
Examples53
Sized enum - HP C Extension53
Declarators55
Syntax55
Description55
Pointer Declarators56
Array Declarators57
Function Declarators58
Type Names60
Syntax60
Description60
Examples60
Type Definitions Using typedef61
Syntax61
Description61
Example62
Initialization63
Syntax63
Description63
Examples65
Function Definitions67
Syntax67
Description67
Examples70
Four-Byte Extended UNIX Code (EUC)71
4. Type Conversions	
Integral Promotions74

Contents

Usual Arithmetic Conversions.	76
Arithmetic Conversions	78
Integral Conversions.	78
Floating Conversions	79
Arrays, Pointers, and Functions.	80
5. Expressions and Operators	
Arithmetic Operators (+, -, *, /, %).	82
Syntax	82
Arguments.	82
Description	83
Array Subscripting ([]).	84
Syntax	84
Description	84
Assignment Operators (=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, =) . . .	86
Syntax	86
Arguments.	87
Description	87
Example.	92
Bit Operators (<<, >>, &, ^, , ~).	93
Syntax	93
Arguments.	93
Description	93
Cast Operator.	99
Syntax	99
Arguments.	99
Description	99
Comma Operator (,)	104
Syntax	104

Contents

Arguments	104
Description	104
Conditional Expression Operator (? :)	105
Syntax	105
Arguments	105
Description	105
Example	106
Function Calls	107
Syntax	107
Description	107
Increment and Decrement Operators	
(++, --)	110
Syntax	110
Arguments	110
Description	110
Examples	113
Logical Operators (&&, , !)	114
Syntax	114
Arguments	114
Description	114
Example	117
Pointer Operators (*, ->, &)	119
Syntax	119
Description	119
Example 1	124
Relational Operators (>, >=, <, ==, !=)	127
Syntax	127
Arguments	127
Description	127
Example	132

Contents

sizeof Operator	133
Syntax	133
Arguments	133
Description	133
Structure and Union Members (., ->)	135
Syntax	135
Description	135
Operator Precedence	136
Precedence among Operators of Same Class	137
Operator Quick Reference	138
Constant Expressions	139
Integral Expressions	140
Floating-Point Expressions	140
lvalue Expressions	141
Pointer Expressions	141
Evaluation of Expressions	142
Examples	142
Evaluation Order of Subexpressions	143
6. Statements	
Branch Statements	146
Syntax	146
Description	146
Examples	146
Compound Statement or Block	148
Syntax	148
Description	148
Example	148

Contents

Expression and Null Statements	149
Syntax	149
Description	149
Example	149
Labeled Statements	150
Syntax	150
Description	150
Example	150
Looping Statements	151
Syntax	151
Description	151
Examples	151
Selection Statements	152
Syntax	152
Description	152
Examples	152
break	153
Syntax	153
Description	153
Example	153
continue	154
Syntax	154
Description	154
Example	154
do...while	156
Syntax	156
Arguments	156
Description	156
Example	157

Contents

for	158
Syntax	158
Arguments.....	158
Description	158
Example.....	160
goto	162
Syntax	162
Arguments.....	162
Description	162
Example.....	162
if.....	164
Syntax	164
Arguments.....	164
Description	164
Example.....	166
return	168
Syntax	168
Arguments.....	168
Description	168
Example.....	170
switch	171
Syntax	171
Arguments.....	171
Description	171
Example.....	173
while	175
Syntax	175
Arguments.....	175
Description	175
Example.....	176

Contents

7. Preprocessing Directives

Overview of the Preprocessor	178
Source File Inclusion (#include)	180
Syntax	180
Description	180
Examples.	181
Macro Replacement (#define, #undef)	182
Syntax	182
Description	182
Predefined Macros	186
Conditional Compilation (#if, #ifdef, ..#endif)	187
Syntax	187
Description	187
Examples.	189
Line Control (#line)	190
Syntax	190
Description	190
Example	190
Pragma Directive (#pragma).	191
Syntax	191
Description	191
Example	191
Error Directive (#error).	192
Syntax	192
Examples.	192
Trigraph Sequences	193

Contents

8. C Library Functions

9. Compiling and Running HP C Programs

Compiling HP C Programs	198
Compatibility Mode vs. ANSI C Mode	198
The cc(1) Command	199
Specifying Files to the cc Command	199
Specifying Options to the cc Command	199
HP C Compiler Options	200
Examples of Compiler Commands	231
Environment Variables	233
Compiling for Different Versions of the PA-RISC Architecture	235
Using +DA to Generate Code for a Specific Version of PA-RISC ...	235
Using +DS to Specify Instruction Scheduling	236
Compiling in Networked Environments	237
Pragmas	238
Initialization and Termination Pragmas	238
Intrinsic Pragmas	240
Copyright Notice and Identification Pragmas	241
Data Alignment Pragmas	242
Listing Pragmas	246
Optimization Pragmas	247
Shared Library Pragma	248
FastCall Pragmas	250
Running HP C Programs	251

10. HP C/HP-UX Implementation Topics

Data Types	254
Bit-Fields	257

Contents

IEEE Floating-Point Format	258
Lexical Elements	260
Structures and Unions	260
Type Mismatches in External Names	261
Expressions	261
Pointers	261
Maximum Number of Dimensions of an Array	262
Scope of extern Declarations	262
Conversions Between Floats, Doubles, and Long Doubles	262
Statements	263
Preprocessor	263
Library Functions and Header Files	264
The Math Library	264
Other Library Functions	264
The varargs Macros	265
Example	265
C9X standard macros	267
Variable names	268
How HP C implements GNU and C9X macros	268
HP Specific Type Qualifiers	270
Location of Files	270
11. Using Intrinsic	
INTRINSIC Pragma	276
Examples	276
INTRINSIC_FILE Pragma	278

Contents

12. The Listing Facility	
Listing Format	280
Compatibility Mode	280
ANSI Mode	280
Listing Pragmas	281
Listing Options	282
Identifier Maps	282
Code Offsets	284
Example	284
A. Syntax Summary	
Lexical Grammar	288
Tokens	288
Keywords	288
Identifiers	289
Constants	289
String Literals	291
Operators	291
Punctuators	292
Header Names	292
Preprocessing Numbers	292
Phrase Structure Grammar	293
Expressions	293
Declarations	295
Statements	297
External Definitions	298
Preprocessing Directives	299
Index	301

Printing History

New editions are complete revisions of the manual. The dates on the title page change only when a new edition is published.

The software code printed alongside the data indicates the version level of the software product at the time the manual or update 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.

Table 1 **Printing History**

First Edition	August 1989	HP-UX: 92453-01A.07.09
Second Edition	January 1991	HP-UX: 92453-01A.08.11
Third Edition	August 1992	HP-UX: 92453-01A.09.17
Fourth Edition	January 1994	HP-UX: 92453-01A.09.61
Fifth Edition	June 1996	HP C/HP-UX A.10.32
Sixth Edition	May 1997	HP C/HP-UX A.10.33
Seventh Edition	November 1997	HP C/HP-UX A.11.00
Eighth Edition	June 1998	HP C/HP-UX A.11.00.13 (HP-UX 11.0 June 1998 Extension Pack)
Ninth Edition	June 2000	HP C/HP-UX B.11.01.20

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

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

About this Manual

This manual presents reference information on the C programming language, as implemented on HP 9000 computers. This manual is intended for experienced C programmers who are familiar with HP computer systems. The following features have changed:

June 2000, Edition 9, part number B3901-90002, documents new HP C features that support C99 industry standards, and new options that result in improved performance of the HP C compiler on HP9000 V- and K-class servers, and workstations.

Conventions

C statements are described generally, and then each statement is covered separately. All syntactic categories are fully defined.

Table 2 **Convention Summary**

NOTATION	DESCRIPTION
(see margin)	Change bars in the margin show where substantial changes have been made to the manual since the last edition.
nonitalics	Within syntax descriptions, nonitalicized words represent literals. Enter them exactly as shown. This includes nonitalicized braces and brackets appearing within syntactic descriptions. Nonitalicized words and punctuation characters appear in computer font. In the following example, you must provide both the keyword and the trailing semicolon: <code>break;</code>

NOTATION	DESCRIPTION
<i>italics</i>	<p>Within syntax descriptions, italicized words denote argument names, program names, or strings that you must replace with an appropriate value. In the following example, you must replace <i>identifier</i> with the name of a label you want the program to transfer execution to at this point:</p> <pre>goto <i>identifier</i>;</pre>
[]	<p>Within syntax descriptions, brackets surround optional elements. For example, the <i>expression</i> in the return statement is optional:</p> <pre>return [<i>expression</i>];</pre>
.	<p>Within examples, vertical ellipses may show where portions of the example were omitted.</p>

Related Documents

Refer to the following materials for further information on C language programming:

American National Standard for Information Systems — Programming Language — C, ANSI/ISO 9899-1990.

COBOL/HP-UX Operating Manual — This manual provides information on calling C subprograms from COBOL programs on HP-UX. It also explains how to call COBOL subprograms from C.

HP-UX 64-bit Porting and Transition Guide — Describes the changes you need to make to compile, link, and run programs in 64-bit mode. This document is also available online at <http://docs.hp.com>, and in the Postscript file `/opt/ansic/newconfig/RelNotes/64bitTrans.bk.ps`.

HP-UX Floating-Point Guide — This manual describes the IEEE floating-point standard, the HP-UX math libraries on HP 9000 systems, performance tuning related to floating-point routines, and floating-point coding techniques that can affect application results.

HP Fortran 90 Programmer's Guide — This manual explains how to call C programs from the HP Fortran 90 compiler on HP-UX.

HP Pascal/HP-UX Programmer's Guide — This manual describes how to call C programs from Pascal on HP-UX systems.

HP-UX Linker and Libraries Online User Guide — This online help describes programming in general on HP-UX. For example, it covers linking, loading, shared libraries, and several other HP-UX programming features.

HP-UX Reference — For HP-UX 11.00 and 10.30 the manpages are available in Instant Information under the title *HP-UX Reference* and via the `man` command. For HP-UX 10.20 the manpages are available in LaserROM and via the `man` command. They document commands, system calls, subroutine libraries, file formats, device files, and other HP-UX related topics.

Parallel Programming Guide for HP-UX Systems — Describes efficient parallel programming techniques available using HP Fortran 90, HP C, and HP aC++ on HP-UX.

1

What is HP C?

HP C originates from the C language designed in 1972 by Dennis Ritchie at Bell Laboratories. It descended from several ALGOL-like languages, most notably BCPL and a language developed by Ken Thompson called B.

C has been called a "low-level, high-level" programming language. C's operators and data types closely match those found in modern computers. The language is concise and C compilers produce highly efficient code. C has traditionally been used for systems programming, but it is being used increasingly for general applications.

The most important feature that C provides is portability. In addition, C provides many facilities such as useful data types, including pointers and strings, and a functional set of data structures, operators, and control statements.

ANSI Mode

Unless you are writing code that must be recompiled on a system where ANSI C is not available, it is recommended that you use the ANSI mode of compilation for your new development. It is also recommended that you use ANSI mode to recompile existing programs after making any necessary changes.

Because an ANSI-conforming compiler is required to do more thorough error detection and reporting than has been traditional among C compilers in the past, you may find that your productivity will be enhanced because more errors will be caught at compile time. This may be especially true if you use function prototypes.

If you do not specify the mode of compilation, beginning with the HP-UX 10.30 operating system release, it defaults to `-Ae`.

Compatibility Mode

You may not want to change your existing code, or you may have old code that relies on certain non-ANSI features. Therefore, a compatibility mode of compilation has been provided. In this mode, virtually all programs that compiled and executed under previous releases of HP C/HP-UX will continue to work as expected.

In HP-UX 10.20 and earlier releases, compatibility mode is the default compilation mode. In HP-UX 10.30 forward, extended ANSI mode (-Ae) is the default.

About HP C/HP-UX

This manual presents ANSI C as the standard version of the C language. Where certain constructs are not available in compatibility mode, or would work differently, it is noted and the differences are described.

HP C/HP-UX, when invoked in ANSI mode, is a conforming implementation of ANSI C, as specified by American National Standard 9899-1990. This manual uses the terminology of that standard and attempts to explain the language defined by that standard, while also documenting the implementation decisions and extensions made in HP C/HP-UX. It is not the intent of this document to replicate the standard. Thus, you are encouraged to refer to the standard for any fine points of the language not covered here.

HP C Online Help

Online help for HP C is available for HP 9000 workstation and server users. HP C Online Help can be accessed from an HTML browser of your choice. It consists of html files that contain the following reference and how-to information:

- What is HP C?
- Program organization
- Compiling & running HP C programs
- Optimizing HP C programs
- Parallel options & pragmas
- Data types & declarations
- Expressions & operators
- Statements
- Preprocessing directives
- Calling other languages
- Programming for portability
- Migrating C programs to HP-UX
- Error message descriptions

Prerequisites for using HP C Online Help

Before you can begin using HP C Online Help, you should review the following display and browser information. Some reconfiguration of your environment variables may be required.

- You must set the `DISPLAY` environment variable to a (graphical mode) value that can accommodate the display of an HTML browser.
- You may set the `BROWSER` environment variable to point to the location of the your HTML browser. If you do not do this, the compiler will automatically run the browser located in `/opt/ns-navgold/bin/netscape` or in `/opt/ns-communicator/netscape`.

[What is HP C?](#)
[HP C Online Help](#)

- You may set the `CROOTDIR` environment variable to specify the root directory of the online help source. If `CROOTDIR` is not set, the URL of the HP C Online Help defaults to
`file:/opt/ansic/html/guide/${LOCALE}/c_index.html`.
This default is based on the assumption that the compiler binaries are located in `/opt/ansic/bin`.

Accessing HP C Online Help

To access the HP C Online Help, you must be logged onto a system where the most recent version of the HP C compiler is installed. Typing the following at the command line invokes an HTML browser, which displays the main HTML index file for the HP C Online Help system:

```
/opt/ansic/bin/cc +help
```

The actual location of the HTML files is

```
file:/${CROOTDIR}/html/guide/${LOCALE}/c_index.html.
```

If the environment variable `CROOTDIR` is not set, the path will be formed relative to the compiler's root directory; this is usually `/opt/ansic`. The previous section contains instructions on how to set `CROOTDIR`.

NOTE

If the browser path set by the `BROWSER` environment variable does not exist, or if the default browser paths

```
/opt/ns-navgold/bin/netscape or
```

```
/opt/ns-communicator/netscape do not exist, then you must set the  
BROWSER environment variable appropriately.
```

X-Motif CDE Help is obsolete

Previous versions of the HP C compiler, when installed in the X-Motif CDE environment, included a CDE version of the HP C Online Help. This, and the accompanying text-based "charhelp" will no longer be updated in future releases of the HP C compiler.

2

Program Organization

This section describes the following topics:

- Lexical Elements
- Declarations
- Constants
- Structuring a C Program

Lexical Elements

C language programs are composed of **lexical elements**. The lexical elements of the C language are characters and white spaces that are grouped together into **tokens**. This section describes the following syntactic objects:

- White Space, Newlines, and Continuation Lines
- Comments
- Identifiers
- Keywords

White Space, Newlines, and Continuation Lines

In C source files, blanks, newlines, vertical tabs, horizontal tabs, and formfeeds are all considered to be white space characters.

The main purpose of white space characters is to format source files so that they are more readable. The compiler ignores white space characters, except when they are used to separate tokens or when they appear within string literals.

The newline character is not treated as white space in preprocessor directives. A newline character is used to terminate preprocessor directives. See “Overview of the Preprocessor” on page 178 for more information.

The line continuation character in C is the backslash (`\`). Use the continuation character at the end of the line when splitting a quoted string or preprocessor directive across one or more lines of source code.

Spreading Source Code Across Multiple Lines

You can split a string or preprocessor directive across one or more lines. To split a string or preprocessor directive, however, you must use the continuation character (`\`) at the end of the line to be split; for example:

```
#define foo_macro(x,y,z) ((x) + (y))\  
                        * ((z) - (x))  
  
printf("This is an very, very, very lengthy and \  
very, very uninteresting string.");
```

Comments

A comment is any series of characters beginning with `/*` and ending with `*/`. The compiler replaces each comment with a single space character.

HP C allows comments to appear anywhere in the source file except within identifiers or string literals. The C language does not support nested comments.

In the following example, a comment follows an assignment statement:

```
average = total / number_of_components; /* Find mean value. */
```

Comments may also span multiple lines, as in:

```
/*  
   This is a  
   multi-line comment.  
*/
```

Identifiers

Identifiers, also called names, can consist of the following:

- Letters (ISO Latin-1 decimal values 65-90 and 97-122)
- Digits
- Dollar sign (\$) (HP C extension)
- Underscore (_)

The first character must be a letter, underscore or a \$ sign. Identifiers that begin with an underscore are generally reserved for system use. The ANSI/ISO standard reserves all names that begin with two underscores or an underscore followed by an uppercase letter for system use.

NOTE

HP C allows the dollar sign (\$) in identifiers, including using the \$ in the first character, as an extension to the language.

Identifiers cannot conflict with reserved Keywords.

Legal Identifiers

```
meters  
green_eggs_and_ham  
system_name  
UPPER_AND_lower_case  
$name      Legal in HP C, but non-standard
```

Illegal Identifiers

```
20_meters  Starts with a digit  
int        The int type is a reserved keyword  
no$#@good Contains illegal characters
```

Length of Identifiers

HP C identifiers are unique up to 256 characters.

The ANSI/ISO standard requires compilers to support names of up to 32 characters for local variables and 6 characters for global variables.

To improve portability, it is a good idea to make your local variable names unique within the first 32 characters, and global variable names unique within the first 6 characters.

Case Sensitivity in Identifiers

In C, identifier names are always case-sensitive. An identifier written in uppercase letters is considered different from the same identifier written in lowercase. For example, the following three identifiers are all unique:

```
kilograms  
KILOGRAMS  
Kilograms
```

Some HP-UX programming languages (such as Pascal and FORTRAN) are case-insensitive. When writing an HP C program that calls routines from these other languages, you must be aware of this difference in sensitivity.

Strings are also case-sensitive. The system recognizes the following two strings as distinct:

```
"THE RAIN IN SPAIN"  
"the rain in spain"
```

Predefined identifier `__func__`

The HP ANSI C compiler defines the predefined identifier `__func__`, as specified in the C9X Standard. This provides an additional means of debugging your code.

The use of the predefined "`__func__`" identifier allows you to use more informative debugging statements to indicate a specific function. This is useful for fatal errors and conditions that produce warnings. `__func__` can also be used within debugging macros in order to keep track of tasks such as the function calling stack, etc.

The `__func__` identifier is implicitly declared by the compiler in the following manner:

```
static const char __func__[ ] = "<function-name>";
```

The declaration is implicitly added immediately after the opening brace of a function which uses the variable `__func__`. The value `<function-name>` is the name of the lexically-enclosing function.

The following code example fragment illustrates the use of the predefined identifier `__func__`.

```
#include <stdio.h>
void varfunc(void)
{
    printf("%s\n", __func__);
    /* ... */
}
```

Each time the `varfunc` function is called, it prints to the standard output stream:

```
varfunc
```

Keywords

HP C supports the list of keywords shown below. You cannot use keywords as identifiers; if you do, the compiler reports an error. You cannot abbreviate a keyword, and you must enter keywords in lowercase letters.

- auto break case char
- const continue default do
- double else enum extern
- float for goto if

Program Organization

Lexical Elements

- int long register return
- short signed sizeof static
- struct switch __thread typedef
- union unsigned void volatile
- while

NOTE

The `const`, `signed`, and `volatile` keywords are part of the ANSI/ISO standard, but not part of the K&R language definition.

auto

Causes the variable to be dynamically allocated and initialized only when the block containing the variable is being executed. This is the default for local variables.

break

See “break” on page 153.

case

An optional element in a `switch` statement. The case label is followed by an integral constant expression and a (;) colon. No two case constant expressions in the same `switch` statement can have the same value. For example:

```
switch (getchar())
{
    case 'r':
    case 'R':
        moveright();
        break;
    ...
}
```

char

The `char` keyword defines an integer type that is 1 byte long.

A `char` type has a minimum value of -128 and a maximum value of 127.

The numeric range for `unsigned char` is 1 byte, with a minimum value of 0 and a maximum value of 255.

const

Specifies that an object of any type must have a constant value throughout the scope of its name. For example:

```
/* declare factor as a constant float */  
const float factor = 2.54;
```

The value of `factor` cannot change after the initialization.

continue

See “continue” on page 154.

default

A keyword used within the `switch` statement to identify the catch-all-else statement. For example:

```
switch (grade){  
    case 'A':  
        printf("Excellent\n");  
        break;  
    default:  
        printf("Invalid grade\n");  
        break;  
}
```

do

See “do...while” on page 156.

double

A 64-bit data type for representing floating-point numbers.

The lower normalized bound is $2.225E-308$. The lower denormalized bound is $4.941E-324$. The upper bound is $1.798E+308$.

Other floating-point types are `float` and `long double`.

else

See “if” on page 164.

enum

See “Enumeration” on page 52.

extern

Used for declarations both within and outside of a function (except for function arguments). Signifies that the object is declared somewhere else.

float

A 32-bit data type for representing floating-point numbers.

The range for `float` is:

- Min: Least normalized: 1.1755E-38 Least denormalized: 1.4013E-45
- Max: 3.4028E+38

Other floating-point types are `double` and `long double`.

for

See “for” on page 158.

goto

See “goto” on page 162.

if

See “if” on page 164.

int

A 32-bit data type for representing whole numbers.

The range for `int` is -2,147,483,648 through 2,147,483,647.

The range for unsigned `int` is 0 through 4,294,967,295.

long

A 32-bit integer data type in the HP-UX 32-bit data model. The range for `long` is -2,147,483,648 through 2,147,483,647. For the HP-UX 64-bit data model, the `long` data type is 64-bits and the range is the same as the `long long` data type.

The `long long` 64-bit data type is supported as an extension to the language when you use the `-Ae` compile-line option.

The range for `long long` is -9,223,372,036,854,775,808 through +9,223,372,036,854,775,807.

register

Indicates to the compiler that the variable is heavily used and may be stored in a register for better performance.

return

See “return” on page 168.

short

A 16-bit integer data type.

The range for `short` is -32,768 through 32,767.

The range for unsigned `short` is 0 through 65,535.

signed

All integer data types are signed by default. The high-order bit is used to indicate whether a value is greater than or less than zero. Use this modifier for better source code readability. The `signed` keyword can be used with these data types:

- `char`
- `int`
- `enum`
- `long`
- `long long`
- `short`

Whether or not `char` is signed or unsigned by default is implementation-defined. The `signed` keyword lets you explicitly declare (in a portable) way a signed `char`.

sizeof

See “sizeof Operator” on page 133.

static

A variable that has memory allocated for it at program startup time. The variable is associated with a single memory location until the end of the program.

struct

See “Structure and Union Tags” on page 49.

switch

See “switch” on page 171.

__thread

Beginning with the HP-UX 10.30 operating system release, this HP-specific keyword defines a thread specific data variable, distinguishing it from other data items that are shared by all threads. With a thread-specific data variable, each thread has its own copy of the data item. These variables eliminate the need to allocate thread-specific data dynamically, thus improving performance.

This keyword is implemented as an HP-specific type qualifier, with the same syntax as `const` and `volatile`, but not the same semantics.

Syntax examples:

```
__thread int var;  
int __thread var;
```

Semantics for the `__thread` keyword: Only variables of static duration can be thread specific. Thread specific data objects can not be initialized. Pointers of static duration that are not thread specific may not be initialized with the address of a thread specific object — assignment is okay. All global variables, thread specific or not, are initialized to zero by the linker implicitly.

Only one declaration, for example,

```
__thread int x;
```

is allowed in one compilation unit that contributes to the program (including libraries linked into the executable). All other declarations must be strictly references:

```
extern __thread int x;
```

Even though `__thread` has the same syntax as a type qualifier, it does not qualify the type, but is a storage class specification for the data object. As such, it is type compatible with non-thread-specific data objects of the same type. That is, a thread specific data `int` is type compatible with an ordinary `int`, (unlike `const` and `volatile` qualified `int`).

Note that use of the `__thread` keyword in a shared library will prevent that shared library from being dynamically loaded (that is, loaded via an explicit call to `shl_load()`).

typedef

See “Typedef Declarations” on page 19.

union

See “Structure and Union Tags” on page 49.

unsigned

A data type modifier that indicates that no sign bit will be used. The data is assumed to contain values greater than or equal to zero. All integer data types are signed by default. The `unsigned` keyword can be used to modify these data types:

- `char`
- `int`
- `enum`
- `long`
- `long long`
- `short`

void

The `void` data type has three important purposes:

- to indicate that a function does not return a value
- to declare a function that takes no arguments
- to allow you to create generic pointers.

Program Organization

Lexical Elements

To indicate that a function does not return a value, you can write a function definition such as:

```
void func(int a, int b)
{
    . . .
}
```

This indicates that the function `func()` does not return a value. Likewise, on the calling side, you declare `func()` as:

```
extern void func(int, int);
```

volatile

Specifies that the value of a variable might change in ways that the compiler cannot predict. If `volatile` is used, the compiler will not perform certain optimizations on that variable.

while

See “while” on page 175.

Declarations

In general, a variable declaration has the following format:

```
[storage_class_specifier] [data_type] variable_name
    [=initial_value];
```

where:

<i>storage_class_specifier</i>	is an optional keyword.
<i>data_type</i>	is one of the data types described in Chapter 3, "Data Types and Declarations."
<i>variable_name</i>	is a legal identifier.
<i>initial_value</i>	is an optional initializer for the variable.

Here are a few sample variable declarations without storage class specifiers or initial values:

```
int age; /* integer variable "age" */
int length, width; /* abbreviated declaration of two
                   variables*/
float ph; /* floating-point variable "ph" */
char a_letter; /* character variable "a_letter" */
int values[10]; /* array of 10 integers named values */
enum days {mon, wed, fri}; /* enumerated variable "days" */
```

Typedef Declarations

The C language allows you to create your own names for data types with the `typedef` keyword. Syntactically, a `typedef` is similar to a variable declaration except that the declaration is preceded by the `typedef` keyword.

A `typedef` declaration may appear anywhere a variable declaration may appear and obeys the same scoping rules as a normal declaration. Once declared, a `typedef` name may be used anywhere that the type is allowed (such as in a declaration, cast operation, or `sizeof` operation). You can write `typedef` names in all uppercase so that they are not confused with variable names.

You may not include an initializer with a `typedef`.

Program Organization

Declarations

The statement:

```
typedef long int FOUR_BYTE_INT;
```

makes the name `FOUR_BYTE_INT` synonymous with `long int`. The following two declarations are now identical:

```
long int j;  
FOUR_BYTE_INT j;
```

Abstract Global Types

Typedefs are useful for abstracting global types that can be used throughout a program, as shown in the following structure and array declaration:

```
typedef struct {  
    char month[4];  
    int day;  
    int year;  
} BIRTHDAY;  
  
typedef char A_LINE[80]; /* A_LINE is an array of  
                        * 80 characters */
```

Improving Portability

Type definitions can be used to compensate for differences in C compilers. For example:

```
#if SMALL_COMPUTER  
    typedef int SHORTINT;  
    typedef long LONGINT;  
#elif  
    BIG_COMPUTER  
    typedef short SHORTINT;  
    typedef int LONGINT;  
#endif
```

This is useful when writing code to run on two computers, a small computer where an `int` is two bytes, and a large computer where an `int` is four bytes. Instead of using `short`, `long`, and `int`, you can use `SHORTINT` and `LONGINT` and be assured that `SHORTINT` is two bytes and `LONGINT` is four bytes regardless of the machine.

Simplifying Complex Declarations

You can use typedefs to simplify complex declarations. For example:

```
typedef float *PTRF, ARRAYF[], FUNCF();
```

This declares three new types called PTRF (a pointer to a float), ARRAYF (an array of floats), and FUNCF (a function returning a float). These typedefs could then be used in declarations such as the following:

```
PTRF x[5]; /* a 5-element array of pointers to floats */
FUNCF z; /* A function returning a float */
```

Using typedefs for Arrays

The following two examples illustrate what can happen when you mix pointers and typedefs that represent arrays. The problem with the program on the left is that ptr points to an array of 80 chars, rather than a single element of a char array. Because of scaling in pointer arithmetic, the increment operator adds 80 bytes, not one byte, to ptr.

wrong	right
<pre>typedef char STR[80]; STR string, *ptr; main() { ptr = string; printf("ptr = %d\n", ptr); ptr++; printf("ptr = %d\n", ptr); } *** Run-Time Results *** ptr = 3997696 ptr = 3997776</pre>	<pre>typedef char STR[80]; STR string; char *ptr; main() { ptr = string; printf("ptr = %d\n", ptr); ptr++; printf("ptr = %d\n", ptr); } *** Run-Time Results *** ptr = 3997696 ptr = 3997697</pre>

Name Spaces

All identifiers (names) in a program fall into one of four name spaces. Names in different name spaces never interfere with each other. That is, you can use the same name for an object in each of the four name spaces without these names affecting one another. The four name spaces are as follows:

Structure, Union, and Enumeration Tags	Tag names that immediately follow these type specifiers: <code>struct</code> , <code>union</code> , and <code>enum</code> . These types are described in “Structure and Union Specifiers” on page 47.
Member Names	Names of members of a structure or union.
Goto Labels	Names that mark the target of a <code>goto</code> statement.
Function, Variable and All Other Names	Any names that are not members of the preceding three classes.

NOTE

The separate name spaces for `goto` labels and for each `struct`, `union`, or `enum` definition are part of the ANSI/ISO standard, but not part of the K&R language definition.

The following example uses the same name, `overuse`, in four different ways:

```
int main(void)
{
    int overuse;          /* normal identifier */
    struct overuse {     /* tag name */
        float overuse; /* member name */
        char *p;
    } x;
    goto overuse;
overuse: overuse = 3; /* label name */
}
```


Structure, Union, and Enum Names

Each `struct`, `union`, or `enum` defines its own name space, so that different declarations can have the same member names without conflict. The following is legal:

```
struct A {  
    int x;  
    float y;  
};  
struct B {  
    int x;  
    float y;  
};
```

The members in `struct A` are distinct from the members in `struct B`.

Macro Names

Macro names *do* interfere with the other four name spaces. Therefore, when you specify a macro name, do not use this name in one of the other four name spaces. For example, the following program fragment is incorrect because it contains a macro named `square` and a label named `square`:

```
#define square(arg)  arg * arg  
  
int main(void)  
{  
    ...  
square:  
    ...  
}
```

Declarations within code

HP C has added the C9x feature which allows you to declare variables and types inside a block of statements. This also allows declaration of new variables or types, such as `expr_1`, as shown in the `for` statement below:

```
for(expr_1;expr_2;expr_3) statement_1
```

This new variable or type declared in `expr_1` can be used in `expr_2`, `expr_3` and `statement_1`.

NOTE

The HP C/ANSI C compiler implementation of declarations with is code is similar to, but not identical to, the C++ implementation of declarations within code.

Constants

There are four types of constants in C:

- Integer Constants
- Floating-Point Constants
- Character Constants
- String Constants

Every constant has two properties: **value** and **type**. For example, the constant 15 has value 15 and type `int`.

Integer Constants

HP C supports three forms of integer constants:

- decimal** One or more digits from 0-9. The constant must not start with a 0.
- octal** One or more digits from 0-7. The constant must begin with 0.
- hexadecimal** One or more hexadecimal digits from 0-9, a-f, or A-F. The constant must begin with 0x or 0X.

An integer constant is a simple number like 12. It is not an integer variable (like `x` or `y`) or an integer expression.

The data type assigned to an integer constant is the first in which it will fit from the list on the right for the constant declaration on the left:

Table 2-1

Constant	Assigned Data Type
decimal (no suffix)	<code>int</code> , <code>long int</code> , <code>unsigned long int</code>
octal or hex (no suffix)	<code>int</code> , <code>unsigned int</code> , <code>long</code> , <code>unsigned long</code>
letter <code>u</code> or <code>U</code> suffix	<code>unsigned int</code> , <code>unsigned long int</code>

Constant	Assigned Data Type
letter l or L suffix	long, unsigned long
both letters u or U and	unsigned long l or L suffix:
letters ll or LL suffix:	long long, unsigned long long
both letters u or U and ll or LL suffix:	unsigned long long

Integer constants may not contain any punctuation such as commas or periods.

Examples of Integer Constants

The following examples show some legal constants in decimal, octal, and hexadecimal form:

Table 2-2

Decimal	Octal	Hexadecimal
3	003	0x3
8	010	0x8
15	017	0xF
16	020	0x10
21	025	0x15
-87	-0127	-0x57
187	0273	0xBB
255	0377	0xff

Floating-Point Constants

A **floating-point constant** is any number that contains a decimal point and/or exponent sign for scientific notation.

The number may be followed by an `f` or `F`, to signify that it is of type `float`, or by an `l` or `L`, to signify that it is of type `long double`. If the number does not have a suffix, it is of type `double` even if it can be accurately represented in four bytes.

If the magnitude of a floating-point constant is too great or too small to be represented in a `double`, the C compiler will substitute a value that can be represented. This substitute value is not always predictable.

You may precede a floating-point constant with the unary plus or minus operator to make its value positive or negative.

Scientific Notation

Scientific notation is a useful shorthand for writing lengthy floating-point values. In scientific notation, a value consists of two parts: a number called the **mantissa** followed by a power of 10 called the **characteristic** (or **exponent**).

The letter `e` or `E`, standing for exponent, is used to separate the two parts.

The floating-point constant `3e2`, for instance, is interpreted as $3 \cdot (10^2)$, or 300. Likewise, the value `-2.5e-4` is interpreted as $-2.5 / (10^4)$, or -0.00025.

Examples of Floating-Point Constants

Here are some examples of legal and illegal floating-point constants.

Table 2-3

Floating-Point Constants

Constant	Legal or Illegal
3.	legal
35	legal — interpreted as an integer.
3.141	legal
3,500.45	illegal — commas are illegal.
.3333333333	legal
4E	illegal — the exponent must be followed by a number
0.3	legal

Constant	Legal or Illegal
-3e2	legal
4e3.6	illegal — the exponent must be an integer
3.0E5	legal
+3.6	legal
0.4E-5	legal

Character Constants

A **character constant** is any printable character or legal escape sequence enclosed in single quotes. A character constant can begin with the letter L to indicate that it is a wide character constant; this notation is ordinarily used for characters in an extended character set. In HP C, an ordinary character constant occupies one byte of storage; a wide character constant occupies the rightmost byte of a 4-byte integer.

The value of a character constant is the integer ISO Latin-1 value of the character. For example, the value of the constant `x` is 120.

Escape Sequences

HP C supports several escape sequences:

Table 2-4

Character Escape Codes

Escape Code	Character	What it Does
<code>\a</code>	Audible alert	Rings the terminal's bell.
<code>\b</code>	Backspace	Moves the cursor back one space.
<code>\f</code>	Formfeed	Moves the cursor to the next logical page.
<code>\n</code>	Newline	Prints a newline.
<code>\r</code>	Carriage return	Prints a carriage return.

Escape Code	Character	What it Does
\t	Horizontal tab	Prints a horizontal tab.
\v	Vertical tab	Prints a vertical tab.
\\	Backslash	Prints a backslash.
\?	Question mark	Prints a question mark.
\'	Single quote	Prints a single quote.
\"	Double quote	Prints a double quote.

The escape sequences for octal and hexadecimal numbers are commonly used to represent characters. For example, if ISO Latin-1 representations are being used, the letter a may be written as \141 or \x61 and Z as \132 or \x5A. This syntax is most frequently used to represent the null character as \0. This is exactly equivalent to the numeric constant zero (0). When you use the octal format, you do not need to include the zero prefix as you would for a normal octal constant.

Multi-Character Constants

Each character in an ordinary character constant takes up one byte of storage; therefore, you can store up to a 4-byte character constant in a 32-bit integer and up to a 2-byte character constant in a 16-bit integer.

For example, the following assignments are legal:

```
{
    char    x;                /* 1-byte integer */
    unsigned short int si;    /* 2-byte integer */
    unsigned long int li;     /* 4-byte integer */

    /* the following two assignments are portable: */
    x = 'j';                 /* 1-byte character constant */
    li = L'j';               /* 4-byte wide char constant */

    /* the following two assignments are not portable,
       and are not recommended: */
    si = 'ef';               /* 2-character constant */
    li = 'abcd';             /* 4-character constant */
}
```

The variable `si` is assigned the value of `e` and `f`, where each character takes up 8 bits of the 16-bit value. The HP C compiler places the last character in the rightmost (least significant) byte. Therefore, the

constant `ef` will have a hexadecimal value of `6566`. Since the order in which bytes are assigned is machine dependent, other machines may reverse the order, assigning `f` to the most significant byte. In that case, the resulting value would be `6665`. For maximum portability, do not use multi-character constants. Use character arrays instead.

String Constants

A **string constant** is any series of printable characters or escape characters enclosed in double quotes. The compiler automatically appends a null character (`\0`) to the end of the string so that the size of the array is one greater than the number of characters in the string. For example,

```
"A short string"
```

becomes an array with 15 elements:

Figure 2-1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A		s	h	o	r	t		s	t	r	i	n	g	\0

Like a character constant, a string constant can begin with the letter `L` to indicate that it is a string constant in an extended character set.

To span a string constant over more than one line, use the backslash character (`\`), also called the continuation character. The following, for instance, is legal:

```
strcpy(string, "This is a very long string that requires more \
than one line");
```

Note that if you indent the second line, the spaces will be part of the string.

The compiler concatenates adjacent string constants. Therefore, you can also span a string constant over one line as shown:

Program Organization

Constants

```
strcpy(string, "This is a very long string that requires more "  
            "than one line");
```

When you indent the second line with this method, the spaces are not part of the string.

The type of a string is array of `char`, and strings obey the same conversion rules as other arrays. Except when a string appears as the operand of `sizeof` or as an initializer, it is converted to a pointer to the first element of the string. Note also that the null string,

```
" "
```

is legal, and contains a single trailing null character.

Structuring a C Program

When you write a C program, you can put all of your source code into one file or spread it across many files. A typical C source file contains some or all of the following components:

- Preprocessor directives
- Variables
- Functions

Example

The following shows how a program can be organized:

```
/* preprocessor directives */
#include <stdio.h>
#define WEIGHTING_FACTOR 0.6

/* global typedef declaration */
typedef float THIRTY_TWO_BIT_REAL;

/* global variable declaration */
THIRTY_TWO_BIT_REAL correction_factor = 1.15;

/* prototype */
float average (float arg1, THIRTY_TWO_BIT_REAL arg2)

/* start of function body */
{
    /* local variable declaration */
    float mean;
    /* assignment statement */
    mean = (arg1 * WEIGHTING_FACTOR) +
           (arg2 * (1.0 - WEIGHTING_FACTOR));
    /* return statement */
    return (mean * correction_factor);
/* end of function body */
}

int main(void)
/* start of function body */
{
    /* local variable declarations */
    float value1, value2, result;
    /* statements */
    printf("Enter two values -- ");
    scanf("%f%f", &value1, &value2);
    result = average(value1, value2);
}
```

Program Organization

Structuring a C Program

```
/* continuation line */
    printf("The weighted average using a correction \
factor of %4.2f is %5.2f\n", correction_factor, result);
/* end of function body */
}
```

3

Data Types and Declarations

In C, as in many other programming languages, you must usually declare identifiers before you can use them.

The declarable entities in C are:

- objects
- functions
- tags and members of structures, unions, and enumerated types
- type definition names

This chapter describes declarations, type specifiers, storage-class specifiers, structure and union specifiers, enumerations, declarators, type names, and initialization. Data types and declarations are defined using Backus-Naur form.

Program Structure

A *translation unit* consists of one or more declarations and function definitions.

Syntax

```
translation-unit ::=  
    external-declaration  
    translation-unit external-declaration
```

```
external-declaration ::=  
    function-definition  
    declaration
```

Description

A C program consists of one or more translation units, each of which can be compiled separately. A translation unit consists of a source file together with any headers and source files included by the `#include` preprocessing directive. Each time the compiler is invoked, it reads a single translation unit and typically produces a *relocatable object file*. A translation unit must contain at least one declaration or function definition.

Declarations

A declaration specifies the attributes of an identifier or a set of identifiers.

Syntax

```
declaration ::=  
    declaration-specifiers [init-declarator-list] ;  
  
declaration-specifiers ::=  
    storage-class-specifier [declaration-specifiers]  
    type-specifier [declaration-specifiers]  
    type-qualifier [declaration-specifiers]  
  
init-declarator-list ::=  
    init-declarator  
    init-declarator-list , init-declarator  
  
init-declarator ::=  
    declarator  
    declarator = initializer
```

Description

Making a declaration does not necessarily reserve storage for the identifiers declared. For example, the declaration of an external data object provides the compiler with the attributes of the object, but the actual storage is allocated in another translation unit.

A declaration consists of a sequence of specifiers that indicate the linkage, storage duration, and the type of the entities that the declarators denote.

You can declare and initialize objects at the same time using the *init-declarator-list* syntax. The *init-declarator-list* is a comma-separated sequence of declarators, each of which may have an initializer.

Function definitions have a slightly different syntax as discussed in “Function Declarators” on page 58. Also, note that it is often valid to define a tag (struct, union, or enum) without actually declaring any objects.

New Declaration features

HP C has added the C9x feature which allows you to declare variables and types inside a block of statements. This also allows declaration of new variables or types, such as `expr_1`, as shown in the for statement below:

```
for(expr_1;expr_2;expr_3) statement_1
```

This new variable or type declared in `expr_1` can be used in `expr_2`, `expr_3` and `statement_1`.

Caveats

The HP C/ANSI C compiler implementation of declarations within code is similar to, but not identical to, the C++ implementation of declarations within code. When specifying declarations within code in the HP C/ANSI C compiler, do not expect the same behavior in HP aC++. For example:

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

Note the lack of a new block opening for the for statement. The C++ compiler accepts this form, with warnings, but the C compiler does not. The difference in the way the stack is handled causes the difference in behavior.

Previously, the C compiler did not emit the source file information for the global typedefs. To correct this, use `-y` option along with `-g` when debug info is generated. You can generate debug information by compiling with `+Oobjdebug`.

Example

```
int main()
{
    int i=5,j;

    j=i*i;
    printf("%d\n",j);

    int k=j;
    /*This is accepted in the new release of HP C*/

    for(struct aa {int a;int b} AA={10,50};AA.a<=AA.b;AA.a++){
        /*This is accepted by the new feature */
        printf("%d\n",AA.a);
    }
}
```

Examples

Valid Declarations:

```
extern int pressure [ ];           /* size will be declared elsewhere */
extern int lines = 66, pages;     /* declares two variables,
initializes the first one        */
static char private_func (float); /* a function taking a float,
returning a char, not known
outside this unit                */
const float pi = 3.14;           /* a constant float, initialized */
const float * const pi_ptr = &pi; /* a constant pointer to a constant
float, initialized with an
address constant                  */
static j1, j2, j3;               /* initialized to zero by default */
typedef struct
{double real, imaginary;} Complex; /* declares a type name          */
Complex impedance = {47000};       /* second member defaults to zero */
enum color {red=1, green, blue};   /* declares an enumeration tag and
three constants                   */
int const short static volatile signed
really_Strange = {sizeof '\?'};    /* pretty mixed up                */
```

Invalid Declarations:

```
int ;                             /* no identifier */
;                                  /* no identifier */
int i; j;                          /* no specifiers for j */
```

Storage-Class Specifiers

A *storage-class specifier* is one of several keywords that determines the duration and linkage of an object.

Syntax

```
storage-class ::=  
    typedef  
    extern  
    static  
    auto  
    register
```

Description

You can use only one storage-class specifier in a declaration.

The `typedef` keyword is listed as a storage-class specifier because it is syntactically similar to one.

The keyword `extern` affects the linkage of a function or object name. If the name has already been declared in a declaration with file scope, the linkage will be the same as in that previous declaration. Otherwise, the name will have external linkage.

The `static` storage-class specifier may appear in declarations of functions or data objects. If used in an external declaration (either a function or a data object), `static` indicates that the name cannot be referenced by other translation units. Using the `static` storage class in this way allows translation units to have collections of local functions and data objects that are not exported to other translation units at link time.

If the `static` storage class is used in a declaration within a function, the value of the variable is preserved between invocations of that function.

The `auto` storage-class specifier is permitted only in the declarations of objects within blocks. An automatic variable is one that exists only while its enclosing block is being executed. Variables declared with the `auto` storage-class are all allocated when a function is entered. `Auto` variables that have `initializes` are initialized when their defining block is entered normally. This means that `auto` variables with `initializes` are not initialized when their declaring block is not entered through the top.

The `register` storage class suggests that the compiler store the variable in a register, if possible. You cannot apply the `&` (address-of) operator to register variables.

If no storage class is specified and the declaration appears in a block, the compiler defaults the storage duration for an object to automatic. If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the `extern` storage-class specifier.

If no storage class is specified and the declaration appears outside of a function, the compiler treats it as an externally visible object with static duration.

Type Specifiers

Type specifiers indicate the format of the storage associated with a given data object or the return type of a function.

Syntax

```
type-specifier ::=  
char  
short  
int  
long  
long long  
unsigned  
signed  
float  
double  
void  
struct-or-union-specifier  
enum-specifier  
typedef-name
```

Description

Most of the type specifiers are single keywords. (Refer to Chapter 10, “HP C/HP-UX Implementation Topics,” on page 253 for sizes of types.) The syntax of the type specifiers permits more types than are actually allowed in the C language. The various combinations of type specifiers that are allowed are shown in Table 3-1. Type specifiers that are equivalent appear together in a box. For example, specifying `unsigned` is equivalent to `unsigned int`. Type specifiers may appear in any order, possibly intermixed with other declaration specifiers.

Table 3-1

C Type Specifiers

void
char
signed char
unsigned char
short, signed short, short int, or signed short int
unsigned short, or unsigned short int

int, signed, signed int, or no type specifiers
unsigned, or unsigned int
long, signed long, long int, or signed long int
long long, signed long long, long long int, or signed long long int
unsigned long, or unsigned long int
unsigned long long, or unsigned long long int
float
double
long double
<i>struct-or-union specifier</i>
<i>enum-specifier</i>
<i>typedef-name</i>

If no type specifier is provided in a declaration, the default type is int.

Floating-point types in C are float (32 bits), double (64 bits), and long double (128 bits).

HP-Specific Type Qualifiers

Syntax

type-qualifier ::= `__thread`

Description

Beginning with the HP-UX 10.30 operating system release, the `__thread` keyword defines a thread-specific data variable, distinguishing it from other data items that are shared by all threads. With a thread-specific data variable, each thread has its own copy of the data item. These variables eliminate the need to allocate thread-specific data dynamically, thus improving performance.

This keyword is implemented as an HP specific type qualifier, with the same syntax as type qualifiers `const` and `volatile`, but not the same semantics.

Syntax examples:

```
__thread int var;  
int __thread var;
```

Semantics: Only variables of static duration can be thread-specific. Thread-specific data objects can not be initialized. Pointers of static duration that are not thread-specific may not be initialized with the address of a thread-specific object — assignment is allowed. All global variables, thread-specific or not, are initialized to zero by the linker implicitly.

Only one declaration, for example,

```
__thread int x;
```

is allowed in one compilation unit that contributes to the program (including libraries linked into the executable). All other declarations must be strictly references:

```
extern __thread int x;
```

Any other redeclarations of this thread-specific `x` will result in a duplicate definition error at link time.

Even though `__thread` has the same syntax as a type qualifier, it does not qualify the type, but is a storage class specification for the data object. As such, it is type compatible with non-thread-specific data objects of the same type. That is, a thread-specific data `int` is type compatible with an ordinary `int`, (unlike `const` and `volatile` qualified `int`).

NOTE

Use of the `__thread` keyword in a shared library will prevent that shared library from being dynamically loaded (that is, loaded via an explicit call to `shl_load`).

Type Qualifiers

Syntax

```
type-qualifier ::= const  
                  volatile
```

Description

This section describes the *type qualifiers* — `volatile` and `const`.

The `volatile` type qualifier directs the compiler not to perform certain optimizations on an object because that object can have its value altered in ways beyond the control of the compiler.

Specifically, when an object's declaration includes the `volatile` type qualifier, optimizations that would delay any references to (or modifications of) the object will not occur across sequence points. A *sequence point* is a point in the execution process when the evaluation of an expression is complete, and all side-effects of previous evaluations have occurred.

The `volatile` type qualifier is useful for controlling access to memory-mapped device registers, as well as for providing reliable access to memory locations used by asynchronous processes.

The `const` type qualifier informs the compiler that the object will not be modified, thereby increasing the optimization opportunities available to the compiler.

An assignment cannot be made to a constant pointer, but an assignment can be made to the object to which it points. An assignment can be made to a pointer to constant data, but not to the object to which it points. In the case of a constant pointer to constant data, an assignment cannot be made to either the pointer, or the object to which it points.

Type qualifiers may be used alone (as the sole declaration-specifier), or in conjunction with type specifiers, including `struct`, `union`, `enum`, and `typedef`. Type qualifiers may also be used in conjunction with storage-class specifiers.

Table 3-2 illustrates various declarations using the `const` and `volatile` type qualifiers.

Table 3-2 **Declarations using const and volatile**

Declaration	Meaning
<code>volatile int vol_int;</code>	Declares a volatile int variable.
<code>const int *ptr_to_const_int;</code> <code>int const *ptr_to_const_int;</code>	Both declare a variable pointer to a constant int.
<code>int *const const_ptr_to_int</code>	Declares a constant pointer to a variable int.
<code>int *volatile vpi, *pi;</code>	Declares two pointers: vpi is a volatile pointer to an int; pi is a pointer to an int.
<code>int const *volatile vpci;</code>	Declares a volatile pointer to a constant int.
<code>const *pci;</code>	Declares a pointer to a constant int. Since no type specifier was given, it defaults to int.

When a type qualifier is used with a variable typed by a typedef name, the qualifier is applied without regard to the contents of the typedef. For example:

```
typedef int *t_ptr_to_int;
volatile t_ptr_to_int vol_ptr_to_int;
```

In the example above, the type of `vol_ptr_to_int` is `volatile t_ptr_to_int`, which becomes volatile pointer to int. If the type `t_ptr_to_int` were substituted directly in the declaration,

```
volatile int * ptr_to_vol_int;
```

the type would be pointer to volatile int.

Type qualifiers apply to objects, not to types. For example:

```
typedef int * t;
const t *volatile p;
```

In the example above, `p` is a volatile pointer to a `const` pointer to int. `volatile` applies to the object `p`, while `const` applies to the object pointed to by `p`. The declaration of `p` can also be written as follows:

```
t const *volatile p;
```

Data Types and Declarations

Type Qualifiers

If an aggregate variable such as a structure is declared volatile, all members of the aggregate are also volatile.

If a pointer to a volatile object is converted to a pointer to a non-volatile type, and the object is referenced by the converted pointer, the behavior is undefined.

Structure and Union Specifiers

A *structure specifier* indicates an aggregate type consisting of a sequence of named members. A *union specifier* defines a type whose members begin at offset zero from the beginning of the union.

Syntax

```
struct-or-union specifier ::=  
    struct-or-union [identifier] { struct-declaration-list }  
    struct-or-union identifier
```

```
struct-or-union ::=  
    struct  
    union
```

```
struct-declaration-list ::=  
    struct-declaration  
    struct-declaration-list struct-declaration
```

```
struct-declaration ::=  
    specifier-qualifier-list struct-declarator-list;
```

```
specifier-qualifier-list ::=  
    type-specifier [specifier-qualifier-list]  
    type-qualifier [specifier-qualifier-list]
```

```
struct-declarator-list ::=  
    struct-declarator  
    struct-declarator-list , struct-declarator
```

```
struct-declarator ::=  
    declarator  
    [declarator] : constant-expression
```

Description

A *structure* is a named collection of members. Each member belongs to a name space associated with the structure. Members in different structures can have the same names but represent different objects.

Members are placed in physical storage in the same order as they are declared in the definition of the structure. A member's offset is the distance from the start of the structure to the beginning of the member. The compiler inserts pad bytes as necessary to insure that members are properly aligned. For example, if a `char` member is followed by a `float` member, one or more pad bytes may be inserted to insure that the `float` member begins on an appropriate boundary.

Data Types and Declarations

Structure and Union Specifiers

The *HP C/HP-UX Programmer's Guide* provides a detailed comparison of storage and alignment on HP computers.

Unions are like structures except that all members of a `union` have a zero offset from the beginning of the union. In other words, the members overlap. Unions are a way to store different type of objects in the same memory location.

A declarator for a member of a structure or `union` may occupy a specified number of bits. This is done by following the declarator with a colon and a constant non-negative integral expression. The value of the expression indicates the number of bits to be used to hold the member. This type of member is called a bit-field. Only integral type specifiers are allowed for bit-field declarators.

In structures, bit-fields are placed into storage locations from the most significant bits to the least significant bits. Bit-fields that follow one another are packed into the same storage words, if possible. If a bit-field will not fit into the current storage location, it is put into the beginning of the next location and the current location is padded with an unnamed field.

A colon followed by an integer constant expression indicates that the compiler should create an unnamed bit-field at that location. In addition, a colon followed by a zero indicates that the current location is full and that the next bit-field should begin at the start of the next storage location.

Although bit-fields are permitted in unions (ANSI mode only), they are just like any other members of the `union` in that they have a zero offset from the beginning of the union. That is, they are not packed into the same word, as in the case of structures. The special cases of unnamed bit-fields and unnamed bit-fields of length zero behave differently with unions; they are simply unnamed members that cannot be assigned to.

The unary address operator (`&`) may not be applied to bit-fields. This implies that there cannot be pointers to bit-fields nor can there be arrays of bit-fields.

Refer to Chapter 10, "HP C/HP-UX Implementation Topics," on page 253 for more information on bit-fields.

Structure and Union Tags

Structures and unions are declared with the `struct` or `union` keyword. You can follow the keywords with a tag that names the structure or union type much the same as an `enum` tag names the enumerated type. (Refer to “Enumeration” on page 52 for information on enumerated types.) Then you can use the tag with the `struct` or `union` keyword to declare variables of that type without re-specifying member declarations. A structure tag occupies a separate name space reserved for tags. Thus, a structure tag may have the same spelling as a structure member or an ordinary identifier. Structure tags also obey the normal block scope associated with identifiers. Another tag of the same spelling in a subordinate block may hide a structure tag in an outer block.

A `struct` or `union` declaration has two parts: the structure body, where the members of the structure are declared (and possibly a tag name associated with them); and a list of declarators (objects with the type of the structure).

Either part of the declaration can be empty. Thus, you can put the structure body declaration in one place, and use the `struct` type in another place to declare objects of that type.

For example, consider the following declarations:

```
struct s1 {  
    int x;  
    float y;  
};  
  
struct s1 obj1, *obj2;
```

The first example declares only the `struct` body and its associated tag name. The second example uses the `struct` tag to declare two objects — `obj1` and `obj2`. They are, respectively, a structure object of type `struct s1` and a pointer object, pointing to an object of type `struct s1`.

This allows you to separate all the `struct` body declarations into one place (for example, a header file) and use the `struct` types elsewhere in the program when declaring objects.

Consider the following example:

```
struct examp {  
    float f;           /* floating member */  
    int i;            /* integer member */  
};                   /* no declaration list */
```

Data Types and Declarations

Structure and Union Specifiers

In this example, the structure tag is `examp` and it is associated with the structure body that contains a single floating-point quantity and an integer quantity. Note that no objects are declared after the definition of the structure's body; only the tag is being defined.

A subsequent declaration may use the defined structure tag:

```
struct examp x, y[100];
```

This example defines two objects using type `struct examp`. The first is a single structure named `x` and the second, `y`, is an array of structures of type `struct examp`.

Another use for structure tags is to write *self-referential* structures. A structure of type `S` may contain a pointer to a structure of type `S` as one of its members. Note that a structure can never have itself as a member because the definition of the structure's content would be recursive. A pointer to a structure is of fixed size, so it may be a member. Structures that contain pointers to themselves are key to most interesting data structures. For example, the following is the definition of a structure that is the node of a binary tree:

```
struct node {
    float data;           /* data stored at the node */
    struct node *left;    /* left subtree */
    struct node *right;   /* right subtree */
};
```

This example defines the shape of a node type of structure. Note that the definition contains two members (`left` and `right`) that are themselves pointers to structures of type `node`.

The C programming rule that all objects must be defined before use is relaxed somewhat for structure tags. A structure can contain a member that is a pointer to an as yet undefined structure. This allows for mutually referential structures:

```
struct s1 { struct s2 *s2p; };
struct s2 { struct s1 *s1p; };
```

In this example, structure `s1` references the structure tag `s2`. When `s1` is declared, `s2` is undefined. This is valid.

Example

```
struct tag1 {
    int m1;
    int   :16; /* unnamed bit-field */
    int m2:16; /* named bit-field; packed into */
              /* same word as previous member */
    int m3, m4;
}; /* empty declarator list */

union tag2 {
    int u1;
    int   :16;
    int u2:16; /* bit-field, starts at offset 0 */
    int u3, u4;
} fudge1, fudge2; /* declarators denoting objects
                  of the union type */

struct tag1 obj1, *obj2; /* use of type "struct tag1",
                        whose body has been declared above */
```

Enumeration

The identifiers in an enumeration list are declared as constants.

Syntax

```
enum-specifier ::=  
    [ type-specifier ] enum [ identifier ] {enumerator-list}  
    [ type-specifier ] enum identifier
```

```
enumerator-list ::=  
    enumerator  
    enumerator-list , enumerator
```

```
enumerator ::=  
    enumeration-constant  
    enumeration-constant = constant-expression
```

```
enumeration-constant ::= identifier
```

Description

The identifiers defined in the enumerator list are enumeration constants of type `int`. As constants, they can appear wherever integer constants are expected. A specific integer value is associated with an enumeration constant by following the constant with an equal sign (=) and a constant expression. If you define the constants without using the equal sign, the first constant will have the value of zero and the second will have the value of one, and so on. If an enumerator is defined with the equal sign followed by a constant expression, that identifier will take on the value specified by the expression. Subsequent identifiers appearing without the equal sign will have values that increase by one for each constant. For example,

```
enum color {red, blue, green=5, violet};
```

defines red as 0, blue as 1, green as 5, and violet as 6.

Enumeration constants share the same name space as ordinary identifiers. They have the same scope as the scope of the enumeration in which they are defined. You can also use the `int` or `long` type specifier to indicate 4-byte enums, even though 4-byte enums are the default.

The identifier in the `enum` declaration behaves like the tags used in structure and union declarations. If the tag has already been declared, you can use the tag as a reference to that enumerated type later in the program.

```
enum color x, y[100];
```

In this example, the `color` enumeration tag declares two objects. The `x` object is a scalar `enum` object, while `y` is an array of 100 `enums`.

An enumeration tag cannot be used before its enumerators are declared.

Examples

```
enum color {RED, GREEN, BLUE};  
enum objectkind {triangle, square=5, circle}; /* circle == 6 */
```

Sized enum - HP C Extension

By default, the HP C compiler on HP 9000 systems allocates four bytes for all enumerated variables. However, if you know that the range of values being assigned to an `enum` variable is small, you can direct the compiler to allocate only one or two bytes by using the `char` or `short` type specifier. If the range is large, you can direct the compiler to allocate eight bytes by using the `long long` type specifier. You can also use the `long` type specifier to indicate 4-byte `enums`, even though this is the default. For example:

```
long long enum bigger_enum {barge, yacht}; /* 8-byte enum type */  
enum default_enum {ERR1, ERR2, ERR3, ERR4}; /* 4-byte enum type */  
long enum big_enum {ST0, ST1, ST2, ST3}; /* 4-byte enum type */  
short enum small_enum {cats, dogs}; /* 2-byte enum type */  
char enum tiny_enum {alpha, beta}; /* 1-byte enum type */
```

When mixed in expressions, `enums` behave exactly as their similarly sized type counterparts do. That is, an `enum` behaves like an `int`, a `long enum` acts like a `long int`, and a `short enum` acts like a `short int`. You will, however, receive a warning message when you mix `enum` variables with integer or floating-point types, or with differently typed `enums`.

Data Types and Declarations
Enumeration

The `sizeof()` function returns the actual storage allocated when called with *enum-specifier*.

NOTE

enumeration-constants will have the same size as the type specified in the enumeration declaration.

```
char enum {a}; /* sizeof(a) returns 1. */
```

Declarators

A *declarator* introduces an identifier and specifies its type, storage class, and scope.

Syntax

```
declarator ::=  
    [pointer] direct-declarator  
  
direct-declarator ::=  
    identifier  
    ( declarator )  
    direct-declarator [ [constant-expression] ]  
    direct-declarator ( parameter-type-list )  
    direct-declarator ( identifier-list )  
  
pointer ::=  
    * [type-qualifier-list ]  
    * [type-qualifier-list ] pointer  
  
type-qualifier-list ::=  
    type-qualifier  
    type-qualifier-list type-qualifier  
  
parameter-type-list ::=  
    parameter-list  
    parameter-list , . . .  
  
parameter-list ::=  
    parameter-declaration  
    parameter-list , parameter-declaration  
  
parameter-declaration ::=  
    declaration-specifiers declarator  
    declaration-specifiers [abstract-declarator]  
  
identifier-list ::=  
    identifier  
    identifier-list , identifier
```

Description

Various special symbols may accompany declarators. Parentheses change operator precedence or specify functions. The asterisk specifies a pointer. Square brackets indicate an array. The *constant-expression* specifies the size of an array.

Data Types and Declarations

Declarators

A declarator specifies one identifier and may supply additional type information. When a construction with the same form as the declarator appears in an expression, it yields an entity of the indicated scope, storage class, and type.

If an identifier appears by itself as a declarator, it has the type indicated by the type specifiers heading the declaration.

Declarator operators have the same precedence and associativity as operators appearing in expressions. Function declarators and array declarators bind more tightly than pointer declarators. You can change the binding of declarator operators using parentheses. For example,

```
int *x[10];
```

is an array of 10 pointers to `ints`. This is because the array declarator binds more tightly than the pointer declarator. The declaration

```
int (*x)[10];
```

is a single pointer to an array of 10 `ints`. The binding order is altered with the use of parentheses.

Pointer Declarators

If `D` is a declarator, and `T` is some combination of type specifiers and storage class specifiers (such as `int`), then the declaration `T *D` declares `D` to be a pointer to type `T`. `D` can be any general declarator of arbitrary complexity. For example, if `D` were declared as a pointer already, the use of a second asterisk indicates that `D` is a pointer to a pointer to `T`.

Some examples:

```
int *pi;          /* pi: Pointer to an int          */
int **ppi;       /* ppi: Pointer to a pointer to an int        */
int *ap[10];     /* ap: Array of 10 pointers to ints          */
int (*pa)[10];  /* pa: Pointer to array of 10 ints          */
int *fp();      /* fp: Function returning pointer to int     */
int (*pf)();    /* pf: Pointer to function returning an int */
```

The binding of `*` (pointer) declarators is of lower precedence than either `[]` (array) or `()` (function) declarators. For this reason, parentheses are required in the declarations of `pa` and `pf`.

Array Declarators

If D is a declarator, and T is some combination of type specifiers and storage class specifiers (such as `int`), then the declaration

```
T D[constant-expression];
```

declares D to be an array of type T .

You declare multidimensional arrays by specifying additional array declarators. For example, a 3 by 5 array of integers is declared as follows:

```
int x[3][5];
```

This notation (correctly) suggests that multidimensional arrays in C are actually arrays of arrays. Note that the `[]` operator groups from left to right. The declarator `x[3][5]` is actually the same as `((x[3])[5])`. This indicates that `x` is an array of three elements each of which is an array of five elements. This is known as *row-major* array storage.

You can omit the *constant-expression* giving the size of an array under certain circumstances. You can omit the first dimension of an array (the dimension that binds most tightly with the identifier) in the following cases:

- If the array is a formal parameter in a function definition.
- If the array declaration contains an initializer.
- If the array declaration has external linkage and the definition (in another translation unit) that actually allocates storage provides the dimension.

Note that the `long long` data type cannot be used to declare an array's size.

Following are examples of array declarations:

```
int x[10];           /* x: Array of 10 integers          */
float y[10][20];    /* y: Matrix of 10x20 floats         */
extern int z[ ];    /* z: External integer array of undefined
                    dimension          */
int a[ ]={2,7,5,9}; /* a: Array of 4 integers            */
int m[ ][3]= {      /* m: Matrix of 2x3 integers         */
    {1,2,7},
    {6,6,6} };

```

Note that an array of type T that is the formal parameter in a function definition has been converted to a pointer to type T . The array name in this case is a modifiable lvalue and can appear as the left operand of an

Data Types and Declarations

Declarators

assignment operator. The following function will clear an array of integers to all zeros. Note that the array name, which is a parameter, must be a modifiable lvalue to be the operand of the ++ operator.

```
void clear(a, n)
int a[];          /* has been converted to int * */
int n;           /* number of array elements to clear */
{
    while(n--)   /* for the entire array */
        *a++ = 0; /* clear each element to zero */
}
```

Function Declarators

If D is a declarator, and T is some combination of type specifiers and storage class specifiers (such as `int`), then the declaration

$T D$ (*parameter-type-list*)

or

$T D$ (*identifier-list*)

declares D to be a function returning type T . A function can return any type of object except an array or a function. However, functions can return pointers to functions or arrays.

If the function declarator uses the form with the *parameter-type-list*, it is said to be in "prototype" form. The parameter type list specifies the types of, and may declare identifiers for, the parameters of the function. If the list terminates with an ellipsis (...), no information about the number of types of the parameters after the comma is supplied. The special case of `void` as the only item in the list specifies that the function has no parameters.

If a function declarator is not part of a function definition, the optional *identifier-list* must be empty.

Function declarators using prototype form are only allowed in ANSI mode.

Functions can also return structures. If a function returns a structure as a result, the called function copies the resulting structure into storage space allocated in the calling function. The length of time required to do the copy is directly related to the size of the structure. If pointers to structures are returned, the execution time is greatly reduced. (But beware of returning a pointer to an `auto struct` — the `struct` will disappear after returning from the function in which it is declared.)

The function declarator is of equal precedence with the array declarator. The declarators group from left to right. The following are examples of function declarators:

```
int f();           /* f:  Function returning an int      */
int *fp();        /* fp: Function returning pointer to an int  */
int (*pf)();      /* pf: Pointer to function returning an int */
int (*apf[])();   /* apf: Array of pointers to functions     */
                 /* returning int */
```

Note that the parentheses alter the binding order in the declarations of `pf` and `apf` in the above examples.

Type Names

A *type name* is syntactically a declaration of an object or a function of a given type that omits the identifier. Type names are often used in cast expressions and as operands of the `sizeof` operator.

Syntax

```
type-name ::=
    specifier-qualifier-list [abstract-declarator]

abstract-declarator ::=
    pointer
    [pointer] direct-abstract-declarator

direct-abstract-declarator
    ( abstract-declarator )
    [direct-abstract-declarator] [ [constant-expression] ]
    [direct-abstract-declarator] ( [parameter-type-list] )
```

Description

Type names are enclosed in parentheses to indicate a cast operation. The destination type is the type named in the cast; the operand is then converted to that type.

A type name is a declaration without the identifier specified. For example, the declaration for an integer is `int i`. If the identifier is omitted, only the integer type `int` remains.

Examples

<code>int</code>	<i>int</i>
<code>int *</code>	<i>Pointer to int</i>
<code>int ()</code>	<i>Function returning an int</i>
<code>int *()</code>	<i>Function returning a pointer to int</i>
<code>int (*)()</code>	<i>Pointer to function returning an int</i>
<code>int [3];</code>	<i>Array of 3 int</i>
<code>int *[3];</code>	<i>Array of 3 pointers to int</i>
<code>int (*)[3];</code>	<i>Pointer to an array of 3 int</i>

The parentheses are necessary to alter the binding order in the cases of pointer to function and pointer to array. This is because function and array declarators have higher precedence than the pointer declarator.

Type Definitions Using typedef

The `typedef` keyword, useful for abbreviating long declarations, allows you to create synonyms for C data types and data type definitions.

Syntax

typedef-name ::= identifier

Description

If you use the storage class `typedef` to declare an identifier, the identifier is a name for the declared type rather than an object of that type. Using `typedef` does not define any objects or storage. The use of a `typedef` does not actually introduce a new type, but instead introduces a synonym for a type that already exists. You can use `typedef` to isolate machine dependencies and thus make your programs more portable from one operating system to another.

For example, the following `typedef` defines a new name for a pointer to an `int`:

```
typedef int *pointer;
```

Instead of the identifier `pointer` actually being a pointer to an `int`, it becomes the name for the pointer to the `int` type. You can use the new name as you would use any other type. For example:

```
pointer p, *ppi;
```

This declares `p` as a pointer to an `int` and `ppi` as a pointer to a pointer to an `int`.

One of the most useful applications of `typedef` is in the definition of structure types. For example:

```
typedef struct {  
    float real;  
    float imaginary;  
} complex;
```

The new type `complex` is now defined. It is a structure with two members, both of which are floating-point numbers. You can now use the `complex` type to declare other objects:

```
complex x, *y, a[100];
```

Data Types and Declarations

Type Definitions Using typedef

This declares `x` as a complex, `y` as a pointer to the complex type and `a` as an array of 100 complex numbers. Note that functions would have to be written to perform complex arithmetic because the definition of the complex type does not alter the operators in C.

Other type specifiers (that is, `void`, `char`, `short`, `int`, `long`, `long long`, `signed`, `unsigned`, `float`, or `double`) cannot be used with a name declared by `typedef`. For example, the following `typedef` usage is illegal:

```
typedef long int li;
.
.
.
unsigned li x;
```

`typedef` identifiers occupy the same name space as ordinary identifiers and follow the same scoping rules.

Structure definitions which are used in `typedef` declarations can also have structure tags. These are still necessary to have self-referential structures and mutually referential structures.

Example

```
typedef unsigned long ULONG; /* ULONG is an unsigned long */
typedef int (*PFI)(int); /* PFI is a pointer to a function */
/* taking an int and returning an int */

ULONG v1; /* equivalent to "unsigned long v1" */
PFI v2; /* equivalent to "int (*v2)(int)" */
```

Initialization

An *initializer* is the part of a declaration that provides the initial values for the objects being declared.

Syntax

```
initializer ::=  
    assignment-expression  
    { initializer-list }  
    { initializer-list , }  
  
initializer-list ::=  
    initializer  
    initializer-list , initializer
```

Description

A declarator may include an initializer that specifies the initial value for the object whose identifier is being declared.

Objects with static storage duration are initialized at load time. Objects with automatic storage duration are initialized at run-time when entering the block that contains the definition of the object. An initialization of such an object is similar to an assignment statement.

You can initialize a `static` object with a constant expression. You can initialize a `static` pointer with the address of any previously declared object of the appropriate type plus or minus a constant.

You can initialize an `auto` scalar object with an expression. The expression is evaluated at run-time, and the resulting value is used to initialize the object.

When initializing a scalar type, you may optionally enclose the initializer in braces. However, they are normally omitted. For example

```
int i = {3};
```

is normally specified as

```
int i = 3;
```

When initializing the members of an aggregate, the initializer is a brace-enclosed list of initializers. In the case of a structure with automatic storage duration, the initializer may be a single expression returning a

Data Types and Declarations

Initialization

type compatible with the structure. If the aggregate contains members that are aggregates, this rule applies recursively, with the following exceptions:

- Inner braces may be optionally omitted.
- Members that are themselves aggregates cannot be initialized with a single expression, even if the aggregate has automatic storage duration.

In ANSI mode, the initializer lists are parsed "top-down;" in compatibility mode, they are parsed "bottom-up." For example,

```
int q [3] [3] [2] = {
    { 1 }
    { 2, 3 }
    { 4, 5, 6 }
};
```

produces the following layout:

ANSI Mode	Compatibility Mode
1 0 0 0 0 0	1 0 2 3 4 5
2 3 0 0 0 0	6 0 0 0 0 0
4 5 6 0 0 0	0 0 0 0 0 0

It is advisable to either fully specify the braces, or fully elide all but the outermost braces, both for readability and ease of migration from compatibility mode to ANSI mode.

Because the compiler counts the number of specified initializers, you do not need to specify the size in array declarations. The compiler counts the initializers and that becomes the size:

```
int x[ ] = {1, 10, 30, 2, 45};
```

This declaration allocates an array of int called `x` with a size of five. The size is not specified in the square brackets; instead, the compiler infers it by counting the initializers.

As a special case, you can initialize an array of characters with a character string literal. If the dimension of the array of characters is not provided, the compiler counts the number of characters in the string literal to determine the size of the array. Note that the terminating `\0` is also counted. For example:

```
char message[ ] = "hello";
```

This example defines an array of characters named `message` that contains six characters. It is identical to the following:

```
char message[ ] = {'h','e','l','l','o','\0'};
```

You can also initialize a pointer to characters with a string literal:

```
char *cp = "hello";
```

This declares the object `cp` as a character pointer initialized to point to the first character of the string “hello”.

It is illegal to specify more initializers in a list than are required to initialize the specified aggregate. The one exception to this rule is the initialization of an array of characters with a string literal.

```
char t[3] = "cat";
```

This initializes the array `t` to contain the characters `c`, `a`, and `t`. The trailing `'\0'` character is ignored.

If there are not enough initializers, the remainder of the aggregate is initialized to zero.

Some more examples include:

```
char *errors[ ] = {  
    "undefined file",  
    "input error",  
    "invalid user"  
};
```

In this example, the array `errors` is an array of pointers to character (strings). The array is initialized with the starting addresses of three strings, which will be interpreted as error messages.

An array with element type compatible with `wchar_t` (unsigned int) may be initialized by a wide string literal, optionally enclosed in braces. Successive characters of the wide string literal initialize the members of the array. This includes the terminating zero-valued character, if there is room or if the array is of unknown size.

Examples

```
wchar_t wide_message[ ]=L"x$$z";
```

You initialize structures as you do any other aggregate:

```
struct{  
    int i;  
    unsigned u:3;  
    unsigned v:5;  
    float f;  
    char *p;  
} s[ ] = {  
    {1, 07, 03, 3.5, "cats eat bats" },  
    {2, 2, 4, 5.0, "she said with a smile"}  
};
```

Data Types and Declarations

Initialization

Note that the object being declared (`s`) is an array of structures without a specified dimension. The compiler counts the initializers to determine the array's dimension. In this case, the presence of two initializers implies that the dimension of `s` is two. You can initialize named bit-fields as you would any other member of the structure.

If the value used to initialize a bit-field is too large, it is truncated to fit in the bit-field.

For example, if the value 11 were used to initialize the 3-bit field `u` above, the actual value of `u` would be 3 (the top bit is discarded).

A struct or union with automatic storage duration can also be initialized with a single expression of the correct type.

```
struct SS { int y; };
extern struct SS g(void);
func()
{
    struct SS z = g();
}
```

When initializing a union, since only one union member can be active at one time, the first member of the union is taken to be the initialized member.

The union initialization is only available in ANSI mode.

```
union {
    int      i;
    float    f;
    unsigned u:5;
} = { 15 };
```

Function Definitions

A *function definition* introduces a new function.

Syntax

function-definition ::=
[*declaration-specifiers*] *declarator* [*declaration-list*] *compound-statement*

Description

A function definition provides the following information about the function:

1. **Type.**

You can specify the return type of the function. If no type is provided, the default return type is `int`. If the function does not return a value, it can be defined as having a return type of `void`. You can declare functions as returning any type except a function or an array. You can, however, define functions that return pointers to functions or pointers to arrays.

2. **Formal parameters.** There are two ways of specifying the type and number of the formal parameters to the function:

- a. A function declarator containing an *identifier list*.

The identifiers are formal parameters to the function. You must include at least one declarator for each declaration in the declaration list of the function. These declarators declare only identifiers from the identifier list of parameters. If a parameter in the identifier list has no matching declaration in the declaration list, the type of the parameter defaults to `int`.

- b. A function declarator containing a *parameter type list* (prototype form).

In this case, the function definition cannot include a declaration list. You must include an identifier in each parameter declaration (not an abstract declarator). The one exception is when the parameter list consists of a single parameter of type `void`; in this case do not use an identifier.

NOTE

Function prototypes can be used only in ANSI mode.

3. **Visibility outside defining translation unit.** A function can be local to the translation unit in which it is defined (if the storage class specifier is `static`). Alternatively, a function can be visible to other translation units (if no storage class is specified, or if the storage class is `extern`).
4. **Body of the function.** You supply the body that executes when the function is called in a single compound statement following the optional *declaration-list*.

Do not confuse definition with declaration, especially in the case of functions. Function definition implies that the above four pieces of information are supplied. Function declaration implies that the function is defined elsewhere.

You can declare formal parameters as structures or unions. When the function is called, the calling function's argument is copied to temporary locations within the called function.

All functions in C may be recursive. They may be directly recursive so the function calls itself or they may be indirectly recursive so a function calls one or more functions which then call the original function. Indirect recursion can extend through any number of layers.

In function definitions that do not use prototypes, any parameters of type `float` are actually passed as `double`, even though they are seen by the body of the function as floats. When such a function is called with a float argument, the float is converted back to float on entry into the function.

NOTE

In compatibility mode, the type of the parameter is silently changed to double, so the reverse conversion does not take place.

In a prototype-style definition, such conversions do not take place, and the float is both passed and accessed in the body as a float.

`char` and `short` parameters to nonprototype-style function definitions are always converted to type `int`. This conversion does not take place in prototype-style definitions.

In either case, arrays of type T are always adjusted to pointer to type T, and functions are adjusted to pointers to functions.

Single dimensioned arrays declared as formal parameters need not have their size specified. If the name of an integer array is `x`, the declaration is as follows:

```
int x[ ];
```

For multidimensional arrays, each dimension must be indicated by a pair of brackets. The size of the first dimension may be left unspecified.

The storage class of formal parameters is implicitly “function parameter.” A further storage class of `register` is accepted.

Examples

The following example shows a function that returns the sum of an array of integers.

```
int total(data, n) /* function type, name, formal list */
int data[ ];      /* parameter declarations */
int n;
{
    auto int sum = 0; /* local, initialized */
    auto int i;      /* loop variable */

    for(i=0; i<n; ++i) /* range over all elements */
        sum += data[i]; /* total the data array */
    return sum; /* return the value */
}
```

This is an example of a function definition without prototypes.

```
int func1 (p1, p2) /* old-style function definition */
int p1, p2; /* parameter declarations */
{ /* function body starts */
    int l1; /* local variables */
    l1 = p1 + p2;
    return l1;
}
```

Here is an example of a function definition using prototypes.

```
char *func2 (void) /* new-style definition */
/* takes no parameters */
{
    /* body */
}

int func3 (int p1, char *p2, ...) /* two declared parameters:
p1 & p2 */
/* "..." specifies more,
undeclared parameters
of unspecified type */
{
    /* body */ /* to access undeclared
parameters here, use the
functions declared in the
<stdarg.h> header file. */
}
```


Four-Byte Extended UNIX Code (EUC)

HP C/HP-UX supports four-byte Extended UNIX Code (EUC) characters in filenames, comments, and string literals.

Data Types and Declarations
Four-Byte Extended UNIX Code (EUC)

4

Type Conversions

The use of different types of data within C programs creates a need for data type conversions. For example, some circumstances that may require a type conversion are when a program assigns one variable to another, when it passes arguments to functions, or when it tries to evaluate an expression containing operands of different types. C performs data conversions in these situations.

- **Assignment** — Assignment operations cause some implicit type conversions. This makes arithmetic operations easier to write. Assigning an integer type variable to a floating type variable causes an automatic conversion from the integer type to the floating type.
- **Function call** — Arguments to functions are implicitly converted following a number of 'widening' conversions. For example, characters are automatically converted to integers when passed as function arguments in the absence of a prototype.
- **Normal conversions** — In preparation for arithmetic or logical operations, the compiler automatically converts from one type to another. Also, if two operands are not of the same type, one or both may be converted to a common type before the operation is performed.
- **Casting** — You can explicitly force a conversion from one type to another using a *cast* operation.
- **Returned values** — Values returned from a function are automatically converted to the function's type. For example, if a function was declared to return a `double` and the return statement has an integer expression, the integer value is automatically converted to a `double`.

Conversions from one type to another do not always cause an actual physical change in representation. Converting a 16-bit `short int` into a 64-bit `double` causes a representational change. Converting a 16-bit signed `short int` to a 16-bit unsigned `short int` does not cause a representational change.

Integral Promotions

Wherever an `int` or an `unsigned int` may be used in an expression, a narrower integral type may also be used. The narrow type will generally be widened by means of a conversion called an *integral promotion*. All ANSI C compilers follow what are called *value preserving rules* for the conversion. In HP C the value preserving integral promotion takes place as follows: a `char`, a `short int`, a bit-field, or their signed or unsigned varieties, are widened to an `int`; all other arithmetic types are unchanged by the integral promotion.

NOTE

Many older compilers, including previous releases of HP C/HP-UX, performed integral promotions in a slightly different way, following *unsigned preserving rules*. In order to avoid “breaking” programs that may rely on this non-ANSI behavior, compatibility mode continues to follow the unsigned preserving rules. Under these rules, the only difference is that unsigned `char` and unsigned `short` are promoted to unsigned `int`, rather than `int`.

In the vast majority of cases, results are the same. However, if the promoted result is used in a context where its sign is significant (such as a division or comparison operation), results can be different between ANSI mode and compatibility mode. The following program shows two expressions that are evaluated differently in the two modes.

```
#include <stdio.h>
main ()
{
    unsigned short us = 1;
    printf ("Quotient = %d\n", -us/2);
    printf ("Comparison = %d\n", us<-1);
}
```

In compatibility mode, as with many pre-ANSI compilers, the results are:

```
Quotient    = 2147483647
Comparison  = 1
```

ANSI C gives the following results:

```
Quotient    = 0
Comparison  = 0
```

To avoid situations where unsigned preserving and value preserving promotion rules yield different results, you could refrain from using an unsigned `char` or unsigned `short` in an expression that is used as an

operand of one of the following operators: `>>`, `/`, `%`, `<`, `<=`, `>`, or `>=`. Or remove the ambiguity by using an explicit cast to specify the conversion you want.

If you enable ANSI migration warnings, the compiler will warn you of situations where differences in the promotion rules might cause different results. See Chapter 9, “Compiling and Running HP C Programs,” on page 197 for information on enabling ANSI migration warnings.

Usual Arithmetic Conversions

In many expressions involving two operands, the operands are converted according to the following rules, known as the usual arithmetic conversions. The common type resulting from the application of these rules is also the type of the result. These rules are applied in the sequence listed below.

1. If either operand is `long double`, the other operand is converted to `long double`.
2. If either operand is `double`, the other operand is converted to `double`.
3. If either operand is `float`, the other operand is converted to `float`.
4. Integral promotions are performed on both operands, and then the rules listed below are followed. These rules are a strict extension of the ANSI “Usual Arithmetic Conversions” rule (Section 3.2.1.5). This extension ensures that integral expressions will involve `long long` only if one of the operands is of type `long long`. For ANSI conforming compilation, the integral promotion rule is as defined in Section 3.2.1.1 of the Standard. For non-ANSI compilation, the unsigned preserving promotion rule is used.
 - a. If either operand is `unsigned long long`, the other operand is converted to `unsigned long long`,
 - b. otherwise, if one operand is `long long`, the other operand is converted to `long long`,
 - c. otherwise, if either operand is `unsigned long int`, the other operand is converted to `unsigned long int`,
 - d. otherwise, if one operand is `long int`, and the other is `unsigned int`, and `long int` can represent all the values of an `unsigned int`, then the `unsigned int` is converted to a `long int`. (If one operand is `long int`, and the other is `unsigned int`, and `long int` can NOT represent all the values of an `unsigned int`, then both operands are converted to `unsigned long int`.)
 - e. If either operand is `long int`, the other operand is converted to `long int`.
 - f. If either operand is `unsigned int`, the other operand is converted to `unsigned int`.

g. Otherwise, both operands have type `int`.

NOTE

In compatibility mode, the rules are slightly different.

Step 1 does not apply, because `long double` is not supported in compatibility mode.

Step 3 does not apply, because in compatibility mode, whenever a `float` appears in an expression, it is immediately converted to a `double`.

In step 4, remember that the integral promotions are performed according to the unsigned preserving rules when compiling in compatibility mode.

Arithmetic Conversions

In general, the goal of conversions between arithmetic types is to maintain the same magnitude within the limits of the precisions involved. A value converted from a less precise type to a more precise type and then back to the original type results in the same value.

Integral Conversions

A particular bit pattern, or *representation*, distinguishes each data object from all others of that type. Data type conversion can involve a change in representation.

When signed integer types are converted to unsigned types of the same length, no change in representation occurs. A `short int` value of -1 is converted to an unsigned `short int` value of 65535.

Likewise, when unsigned integer types are converted to signed types of the same length, no representational change occurs. An unsigned `short int` value of 65535 converted to a `short int` has a value of -1.

If a `signed int` type is converted to an unsigned type that is wider, the conversion takes (conceptually) two steps. First, the source type is converted to a signed type with the same length as the destination type. (This involves sign extension.) Second, the resulting signed type is converted to unsigned. The second step requires no change in representation.

If an unsigned integer type is converted to a signed integer type that is wider, the unsigned source type is padded with zeros on the left and increased to the size of the signed destination type.

When a `long long` (or `long` in the 64-bit data model) is converted into another integral data type that is of shorter length, truncation may occur. When a `long long` is converted into a double type, no overflow will occur, but there may be a loss of precision.

In general, conversions from wide integer types to narrow integer types discard high-order bits. Overflows are not detected. Conversions from narrow integer types to wide integer types pad on the left with either zeros or the sign bit of the source type as described above.

A “plain” `char` is treated as signed. A “plain” `int` bit-field is treated as signed.

Floating Conversions

When an integer value is converted to a floating type, the result is the equivalent floating-point value. If it cannot be represented exactly, the result is the nearest representable value. If the two nearest representable values are equally near, the result is the one whose least significant bit is zero.

When a `long long` is converted into a floating type no overflow will occur but may result in loss of precision. Converting a `long long` into a quad precision floating point value should be precise with no overflow.

When a floating type is converted into a `long long` type, the fractional part is discarded and overflow may occur.

When floating-point types are converted to integral types, the source type value must be in the representable range of the destination type or the result is undefined. The result is the whole number part of the floating-point value with the fractional part discarded as shown in the following examples:

```
int i;
i = 9.99;          /* i gets the value 9 */
i = -9.99;        /* i gets the value -9 */

float x1 = 1e38;   /* legal; double is converted to float */
float x2 = 1e39;   /* illegal; value is outside of range
                  for float */

long double x3 = 1.f; /* legal; float is converted to long
                    double */
```

When a `long double` value is converted to a `double` or `float` value, or a `double` value is converted to a `float` value, if the original value is within the range of values representable in the new type, the result is the nearest representable value (if it cannot be represented exactly). If the two nearest representable values are equally near, the result is the one whose least significant bit is zero. When a `float` value is converted to a `double` or `long double` value, or a `double` value is converted to a `long double` value, the value is unchanged.

Arrays, Pointers, and Functions

An expression that has function type is called a *function designator*. For example, a function name is a function designator. With two exceptions, a function designator with type “function returning type” is converted to an expression with type “pointer to function returning type.” The exceptions are when the function designator is the operand of `sizeof` (which is illegal) and when it is the operand of the unary `&` operator.

In most cases, when an expression with array type is used, it is automatically converted to a pointer to the first element of the array. As a result, array names and pointers are often used interchangeably in C. This automatic conversion is not performed in the following contexts: (1) when the array is the operand of `sizeof` or the unary `&`; (2) it is a character string literal initializing an array of characters; or (3) it is a wide string literal initializing an array of wide characters.

5

Expressions and Operators

Operators are grouped as follows:

- Arithmetic Operators (+, -, *, /, %)
- Array Subscripting ([])
- Assignment Operators (=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=)
- Bit Operators (<<, >>, &, ^, |, ~)
- Cast Operator
- Comma Operator (,)
- Conditional Expression Operator (?:)
- Function Calls
- Increment and Decrement Operators (++ , --)
- Logical Operators (&&, | |, !)
- Pointer Operators (*, ->, &)
- Relational Operators (>, >=, <, ==, !=)
- sizeof Operator
- Structure and Union Members (., ->)

See Also:

- Operator Precedence
- Operator Quick Reference

The different types of **expressions** are:

- Constant Expressions
- Integral Expressions
- Floating-Point Expressions
- Pointer Expressions
- lvalue Expressions

Arithmetic Operators (+, -, *, /, %)

Syntax

$exp1 + exp2$

Adds $exp1$ and $exp2$. An exp can be any integer or floating-point expression.

$exp1 - exp2$

Subtracts $exp2$ from $exp1$.

$exp1 * exp2$

Multiplies $exp1$ by $exp2$.

$exp1 / exp2$

Divides $exp1$ by $exp2$.

$exp1 \% exp2$

Finds modulo of $exp1$ divided by $exp2$. (That is, finds the remainder of an integer division.) An expression can be any integer expression.

$-exp$

Negates the value of exp .

$+exp$

Identity (unary plus).

Arguments

exp

Any constant or variable expression.

Description

The addition, subtraction, and multiplication (+, -, and *) operators perform the usual arithmetic operations in C programs. All of the arithmetic operators (except the unary plus and minus operators) bind from left to right. The operands may be any integral or floating-point value, with the following exception:

The modulo operator (%) accepts only integer operands.

The unary plus operator (+*exp*) and the addition and subtraction operators accept integer, floating-point, or pointer operands.

The addition and subtraction operators also accept pointer types as operands. Pointer arithmetic is described in “Pointer Operators (*, -, &)” on page 119.

C's modulo operator (%) produces the remainder of integer division, which equals 0 if the right operand divides the left operand exactly. This operator can be useful for tasks such as determining whether or not a year is a U.S. presidential election year. For example:

```
if (year % 4 == 0)
    printf("This is an Olympic Games year.\n");
else
    printf("There will be no Olympic Games this year.\n");
```

As required by the ANSI/ISO C standard, HP C supports the following relationship between the remainder and division operators:

a equals $a\%b + (a/b) * b$ for any integer values of a and b

The result of a division or modulo division is undefined if the right operand is 0.

The sign reversal or unary minus operator (-) multiplies its sole operand by -1. For example, if x is an integer with the value -8, then -x evaluates to 8.

The result of the identity or unary plus operator (+) is simply the value of the operand.

Refer to “Operator Precedence” on page 136 for information about how these and other operators evaluate with respect to each other.

Array Subscripting ([])

A postfix expression followed by the [] operator is a subscripted reference to a single element in an array.

Syntax

postfix-expression [*expression*]

Description

One of the operands of the subscript operator must be of type pointer to T (T is an object type), the other of integral type. The resulting type is T.

The [] operator is defined so that $E1[E2]$ is identical to $(*((E1)+(E2)))$ in every respect. This leads to the (counterintuitive) conclusion that the [] operator is commutative. The expression $E1[E2]$ is identical to $E2[E1]$.

C's subscripts run from 0 to $n-1$ where n is the array size.

Multidimensional arrays are represented as arrays of arrays. For this reason, the notation is to add subscript operators, not to put multiple expressions within a single set of brackets. For example, `int x[3][5]` is actually a declaration for an array of three objects. Each object is, in turn, an array of five int. Because of this, all of the following expressions are correct:

```
x
x[i]
x[i][j]
```

The first expression refers to the 3 by 5 array of int. The second refers to an array of five int, and the last expression refers to a single int.

The expression `x[y]` is an lvalue.

There is no arbitrary limit on the number of dimensions that you can declare in an array.

Because of the design of multidimensional C arrays, the individual data objects must be stored in row-major order.

As another example, the expression

```
a[i,j] = 0
```

looks as if array `a` were doubly subscripted, when actually the comma in the subscript indicates that the value of `i` should be discarded and that `j` is the subscript into the `a` array.

Assignment Operators (=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=)

Syntax

Ivalue = expression

Simple assignment.

Ivalue += expression

Addition and assignment.

Ivalue -= expression

Subtraction and assignment.

*Ivalue *= expression*

Multiplication and assignment.

Ivalue /= expression

Division and assignment.

Ivalue %= expression

Modulo division and assignment.

Ivalue <<= expression

Left shift and assignment.

Ivalue >>= expression

Right shift and assignment.

Ivalue &= expression

Bitwise AND and assignment.

Ivalue ^= expression

Bitwise XOR and assignment.

Ivalue |= expression

Bitwise OR and assignment.

Arguments

lvalue

Any expression that refers to a region of storage that can be manipulated.

expression

Any legal expression.

Description

The assignment operators assign new values to variables. The equal sign (=) is the fundamental assignment operator in C. The other assignment operators provide shorthand ways to represent common variable assignments.

The Assignment Operator (=)

When the compiler encounters an equal sign, it processes the statement on the right side of the sign and assigns the result to the variable on the left side. For example:

```
x = 3;      /* assigns the value 3 to variable x */
x = y;      /* assigns the value of y to x */
x = (y*z); /* performs the multiplication and
           assigns the result to x */
```

An assignment expression itself has a value, which is the same value that is assigned to the left operand.

The assignment operator has right-to-left associativity, so the expression

```
a = b = c = d = 1;
```

is interpreted as

```
(a = (b = (c = (d = 1))));
```

First 1 is assigned to d, then d is assigned to c, then c is assigned to b, and finally, b is assigned to a. The value of the entire expression is 1. This is a convenient syntax for assigning the same value to more than one variable. However, each assignment may cause quiet conversions, so that

```
int j;
double f;
f = j = 3.5;
```

assigns the truncated value 3 to both f and j. Conversely,

Expressions and Operators

Assignment Operators (=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=)

```
j = f = 3.5;
assigns 3.5 to f and 3 to j.
```

The Other Assignment Operators

C's assignment operators provide a handy way to avoid some keystrokes. Any statement in which the left side of the equation is repeated on the right is a candidate for an assignment operator. If you have a statement like this:

```
i = i + 10;
```

you can use the assignment operator format to shorten the statement to

```
i += 10;
```

In other words, any statement of the form

```
var = var op exp; /* traditional form */
```

can be represented in the following shorthand form:

```
var op = exp; /* shorthand form */
```

where `var` is a variable, `op` is a binary operator, and `exp` is an expression.

The only internal difference between the two forms is that `var` is evaluated only once in the shorthand form. Most of the time this is not important; however, it is important when the left operand has side effects, as in the following example:

```
int *ip;
*ip++ += 1; /* These two statements produce */
*ip++ = *ip++ + 1; /* different results. */
```

The second statement is ambiguous because C does not specify which assignment operand is evaluated first. See “Operator Precedence” on page 136 for more information concerning order of evaluation.

Assignment Type Conversions

Whenever you assign a value to a variable, the value is converted to the variable's data type if possible. In the example below, for instance, the floating-point constant 3.5 is converted to an `int` so that `i` gets the integer value 3.

```
int i;
i = 3.5;
```

Integer to Character Conversions . Unlike arithmetic conversions, which always expand the expression, assignment conversions can truncate the expression and therefore affect its value. For example, suppose `c` is a `char`, and you make the following assignment:

```
c = 882;
```

The binary representation of 882 is

```
00000011 01110010
```

This number requires two bytes of storage, but the variable `c` has only one byte allocated for it, so the two upper bits don't get assigned to `c`. This is known as **overflow**, and the result is not defined by the ANSI/ISO C standard or the K&R language definition for `signed` types. HP C simply ignores the extra byte, so `c` would be assigned the rightmost byte:

```
01110010
```

This would erroneously give `c` the value of 114. The principle illustrated for `chars` also applies to `shorts`, `ints`, and `long ints`. For `unsigned` types, however, C has well-defined rules for dealing with overflow conditions. When an integer value `x` is converted to a smaller `unsigned integer` type, the result is the non-negative remainder of

$$x / (U_MAX+1)$$

where `U_MAX` is the largest number that can be represented in the shorter `unsigned` type. For example, if `j` is an `unsigned short`, which is two bytes, then the assignment

```
j = 71124;
```

assigns to `j` the remainder of

$$71124 / (65535+1)$$

The remainder is 5588. For non-negative numbers, and for negative numbers represented in two's complement notation, this is the same result that you would obtain by ignoring the extra bytes.

Integer to Float Conversions . You may assign an integer value to a floating-point variable. In this case, the integer value is implicitly converted to a floating-point type. If the floating-point type is capable of representing the integer, there is no change in value. If `f` is a `double`, the assignment

```
f = 10;
```

Expressions and Operators

Assignment Operators (=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=)

is executed as if it had been written

```
f = 10.0;
```

This conversion is invisible. There are cases, however, where a floating-point type is not capable of exactly representing all integer values. Even though the range of floating-point values is generally greater than the range of integer values, the precision may not be as good for large numbers. In these instances, conversion of an integer to a floating-point value may result in a loss of precision. Consider the following example:

```
#include <stdio.h>
int main(void)
{
    long int j = 2147483600;
    float x;
    x = j;
    printf("j is %d\nx is %10f\n", j, x);
    exit(0);
}
```

If you compile and execute this program, you get:

```
j is 2147483600
x is 2147483648.000000
```

Float to Integer Conversions . The most risky mixture of integer and floating-point values is the case where a floating-point value is assigned to an integer variable. First, the fractional part is discarded. Then, if the resulting integer can fit in the integer variable, the assignment is made. In the following statement, assuming `j` is an `int`, the double value 2.5 is converted to the `int` value 2 before it is assigned.

```
j = 2.5;
```

This causes a loss of precision which could have a dramatic impact on your program. The same truncation process occurs for negative values. After the assignment

```
j = -5.8;
```

the value of `j` is -5.

An equally serious situation occurs when the floating-point value cannot fit into an integer. For example:

```
j = 999999999999.0
```

This causes an overflow condition which will produce unpredictable results. As a general rule, it is a good idea to keep floating-point and integer values separate unless you have a good reason for mixing them.

Double to Float Conversions . As is the case with assigning floating-point values to integer variables, there are also potential problems when assigning double values to float variables. There are two potential problems: loss of precision and an overflow condition. In HP C a double can represent approximately 16 decimal places, and a float can only represent 7 decimal places. If `f` is a float variable, and you make the assignment

```
f = 1.0123456789
```

the computer rounds the double constant value before assigning it to `f`. The value actually assigned to `f`, therefore, will be 1.012346 (in double-to-float conversions, HP C always rounds to the nearest float value). The following example shows rounding due to conversions.

```
/* Program name is "float_rounding". It shows how double values
   can be rounded when they are assigned to a float. */
#include <stdio.h>
int main(void)
{
    float f32;
    double f64;
    int i;
    for (i = 1, f64 = 0; i < 1000; ++i)
        f64 += 1.0/i;
    f32 = f64;
    printf("Value of f64: %1.7f\n", f64);
    printf("Value of f32: %1.7f\n", f32);
}
```

The output is

```
Value of f64: 7.4844709
Value of f32: 7.4844708
```

Floating-Point Overflows . A serious problem occurs when the value being assigned is too large to be represented in the variable. For example, the largest positive number that can be represented by a float is approximately $3e38$. However, neither the K&R language definition nor the ANSI/ISO C standard defines what happens if you try to make an assignment outside this range. Suppose, for example, that your program contains the following assignment:

```
f = 2e40;
```

In this simple case, the compiler recognizes the problem and reports a compile-time error. In other instances, however, a run-time error could result.

Expressions and Operators

Assignment Operators (=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=)

Example

```
/* Following are examples of the use of each
 * assignment operator. In each case, x = 5
 * and y = 2 before the statement is executed. */

x = y;          x = 2
x += y + 1;    x = 8
x -= y * 3;    x = -1
x *= y + 1;    x = 15
x /= y;        x = 2
x %= y;        x = 1
x <<= y;       x = 20
x >>= y;       x = 1
x &= y;        x = 0
x ^= y;        x = 7
x |= y;        x = 7
x = y = 1;    x = 1, y = 1
```

Bit Operators (<<, >>, &, ^, |, ~)

Syntax

exp1 << exp2

Left shifts (logical shift) the bits in *exp1* by *exp2* positions.

exp1 >> exp2

Right shifts (logical or arithmetic shift) the bits in *exp1* by *exp2* positions.

exp1 & exp2

Performs a bitwise AND operation.

exp1 ^ exp2

Performs a bitwise OR operation.

exp1 | exp2

Performs a bitwise inclusive OR operation.

~exp1

Performs a bitwise negation (one's complement) operation.

Arguments

exp1 and *exp2*

Any integer expression.

Description

The bit operators access specific bits in an object. HP C supports the usual six bit operators, which can be grouped into shift operators and logical operators.

Bit-Shift Operators

The << and >> operators shift an integer left or right respectively. The operands must have integer type, and all automatic promotions are performed for each operand. For example, the program fragment

Expressions and Operators

Bit Operators (<<, >>, &, ^, |, ~)

```
short int  to_the_left = 53, to_the_right = 53;
short int  left_shifted_result, right_shifted_result;

left_shifted_result = to_the_left << 2;
right_shifted_result = to_the_right >> 2;
```

sets `left_shifted_result` to 212 and `right_shifted_result` to 13.
The results are clearer in binary:

```
      base 2          base 10
0000000000110101      53
0000000011010100     212 /* 53 shifted left 2 bits */
0000000000001101      13 /* 53 shifted right 2 bits */
```

Shifting to the left is equivalent to multiplying by powers of two:

`x << y` is equivalent to `x * 2y`.

Shifting non-negative integers to the right is equivalent to dividing by powers of 2:

`x >> y` is equivalent to `x / 2y`.

The `<<` operator always fills the vacated rightmost bits with zeros. If *exp1* is unsigned, the `>>` operator fills the vacated leftmost bits with zeros. If *exp1* is signed, then `>>` fills the leftmost bits with ones (if the sign bit is 1) or zeros (if the sign bit is 0). In other words, if *exp1* is signed, the two bit-shift operators preserve its sign.

Not all compilers preserve the sign bit when doing bit-shift operations on signed integers. The K&R language definition and the ANSI standard make this behavior implementation-defined.

Make sure that the right operand is not larger than the size of the object being shifted. For example, the following produces unpredictable and nonportable results because ints have fewer than 50 bits:

```
10 >> 50
```

You will also get nonportable results if the shift count (the second operand) is a negative value.

Bit Logical Operators

The logical bitwise operators are similar to the Boolean operators, except that they operate on every bit in the operand(s). For instance, the bitwise AND operator (&) compares each bit of the left operand to the

corresponding bit in the righthand operand. If both bits are 1, a 1 is placed at that bit position in the result. Otherwise, a 0 is placed at that bit position.

Bitwise AND (&) Operator . The bitwise AND operator performs logical operations on a bit-by-bit level using the following truth table:

Table 5-1

Truth Table for the bitwise AND operator, (&)

bit x of op1	bit x of op2	bit x of result
0	0	0
0	1	0
1	0	0
1	1	1

The following table shows an example of the bitwise AND operator:

Table 5-2

The Bitwise AND Operator

Expression	Hexadecimal Value	Binary Representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 & 5722	0x0452	00000100 01010010

Bitwise Inclusive (|) OR . The bitwise inclusive OR operator performs logical operations on a bit-by-bit level using the following truth table:

Table 5-3 Truth Table for the inclusive OR operator, (|)

bit x of op1	bit x of op2	bit x of result
0	0	0
0	1	1
1	0	1
1	1	1

The bitwise inclusive OR operator (|) places a 1 in the resulting value's bit position if either operand has a bit set at the position.

Table 5-4 Example Using the Bitwise Inclusive OR Operator

Expression	Hexadecimal Value	Binary Representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 5722	0x36DE	00110110 11011110

Bitwise exclusive OR (^) . The bitwise exclusive OR operator performs logical operations on a bit-by-bit level using the following truth table:

Table 5-5 Truth Table for the exclusive OR, ^

bit x of op1	bit x of op2	bit x of result
0	0	0
0	1	1
1	0	1
1	1	0

The bitwise exclusive OR (XOR) operator (^) sets a bit in the resulting value's bit position if either operand (but not both) has a bit set at the position.

Table 5-6 Example Using the XOR Operator

Expression	Hexadecimal Value	Binary Representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 ^ 5722	0x328C	00110010 10001100

Bitwise Complement (~) . The bitwise complement operator (~) performs logical operations on a bit-by-bit level using the following truth table:

Table 5-7 Truth table for the ~, Bitwise Complement

bit x of op2	result
0	1
1	0

The bitwise complement operator (~) reverses each bit in the operand:

Table 5-8**Example Using the Bitwise Complement Operator**

Expression	Hexadecimal Value	Binary Representation
9430	0x24d6	00100100 11010110
~9430	0xdb29	11011011 00101001

Cast Operator

Syntax

(data_type) exp

Arguments

data_type

Any scalar data type, including a scalar data type created through a *typedef*. The *data_type* cannot be an aggregate type, but it can be a pointer to an aggregate type.

exp

Any scalar expression.

Description

To cast a value means to explicitly convert it to another data type. For example, given the two definitions:

```
int    y = 5;
float  x;
```

The following cast operation casts the value of *y* to float:

```
x = (float) y; /* x now equals 5.0 */
```

Here are four more casts (assume that *j* is a scalar data type):

```
i = (float) j; /* Cast j's value to float */
i = (char *)j; /* Cast j's value to a pointer to a char */
i = ((int *)())j; /* Cast j's value to a pointer to a function
                  returning an int */
i = (float) (double) j; /* Cast j's value first to a double
                       and then to a float */
```

It is important to note that if *exp* is a variable, a cast does not change this variable's data type; it only changes the type of the variable's value for that one expression. For instance, in the preceding casting examples, the cast does not produce any permanent effect on variable *j*.

There are no restrictions on casting from one scalar data type to another, except that you may not cast a void object to any other type. You should be careful when casting integers to pointers. If the integer value does not represent a valid address, the results are unpredictable.

A cast expression may not be an lvalue.

Casting Integers to Other Integers

It is possible to cast one integer into an integer of a different size and to convert a floating-point value, enumeration value or pointer to an integer. Conversions from one type of integer to another fall into five cases (A-E) as shown:

Table 5-9 Integer Conversions

Original Type	char	short	int	unsigned char	unsigned short	unsigned int
char	A	B	B	D	E	E
short	C	A	B	C	D	E
int (long)*	C	C	A*	C	C	D*
unsigned char	D	B	B	A	B	B
unsigned short	C	D	B	C	A	B
unsigned int	C	C	D	C	C	A

* Case C for long in 64-bit mode.

CASE A: Trivial Conversions . It is legal to "convert" a value to its current type by casting it, but this conversion has no effect.

CASE B: Integer Widening . Casting an integer to a larger size is fairly straightforward. The value remains the same, but the storage area is widened. The compiler preserves the sign of the original value by filling the new leftmost bits with ones if the value is negative, or with zeros if the value is positive. When it converts to an unsigned integer, the value is always positive, so the new bits are always filled with zeros. The following table illustrates this principle.

```

char i =           hex    dec
                37      55
(short) i =>      0037    55

```

```
(int) i      => 00000037      55

char j =          c3      -61
(short) j =>      ffc3      -61
(int) j   =>      ffffffff    -61

unsigned char k =  37      55
(short) k   =>     0037      55
(int) k     =>     00000037    55
```

CASE C: Casting Integers to a Smaller Type . When an `int` value is cast to a narrower type (`short` or `char`), the excess bits on the left are discarded. The same is true when a `short` is cast to a `char`, or when a `long` in 64-bit mode is cast to an `int`. For instance, if an `int` is cast to a `short`, the 16 leftmost bits are truncated. The following table of values illustrates these conversions.

	hex	dec
signed long int i =	cf34bf1	217271281
(signed short int)i =>	4bf1	19441
(signed char)i =>	f1	-15
(unsigned char)i =>	f1	241

If, after casting to a signed type, the leftmost bit is 1, then the number is negative. However, if you cast to an unsigned type and after the shortening the leftmost bit is 1, then that 1 is part of the value (it is not the sign bit).

CASE D: Casting from Signed to Unsigned, and Vice Versa . When the original type and the converted type are the same size, a representation change is necessary. That is, the internal representation of the value remains the same, but the sign bit is interpreted differently by the compiler. For instance:

	hex	dec	hex	dec
signed int i =	ffffffca9	-855	0000f2a1	62113
(unsigned int)i =>	ffffffca9	4294966441	0000f2a1	62113

The hexadecimal notation shows that the numbers are the same internally, but the decimal notation shows that the compiler interprets them differently.

CASE E: Casting Signed to Unsigned and Widening . This case is equivalent to performing two conversions in succession. First, the value is converted to the signed widened type as described in case B, and then it is converted to unsigned as described in case D. In the following assignments, the new leftmost bits are filled with ones to preserve negativeness even though the final value is unsigned.

Expressions and Operators

Cast Operator

```
signed short int  i =          hex      dec
                    ff55          -171
(unsigned long int)i =>  fffff55  4294967125
```

Casting Floating-Point Values to Integers . Casting floating-point values to integers may produce useless values if an overflow condition occurs. The conversion is made simply by truncating the fractional part of the number. For example, the floating-point value 3.712 is converted to the integer 3, and the floating-point value -504.2 is converted to -504.

Here are some more examples:

```
float f = 3.700, f2 = -502.2, f3 = 7.35e9;

(int)f          => 3
(unsigned int)f => 3
(char)f         => 3

(int)f2         => -502          in decimal    fffffe0a in hex
(unsigned int)f2 => 4294966794  in decimal or fffffe0a in hex
(char)f2        => 10          in decimal    0a in hex

(int)f3         => run-time error
(unsigned int)f3 => run-time error
(char)f3        => run-time error
```

Converting a large `float` to a `char` produces unpredictable results if the rounded value cannot fit in one byte. If the value cannot fit in four bytes, the run-time system issues an overflow error.

Casting Enumerated Values to Integers

When you cast an enumerated expression, the conversion is performed in two steps. First, the enumerated value is converted to an `int`, and then the `int` is converted to the final target data type. The sign is preserved during these conversions.

Casting Double to Float and Vice Versa

When you cast a `float` to a `double`, the system extends the number's precision without changing its true value. However, when you cast a `double` to a `float`, the system shrinks the number's precision, and this shrinking may change the number's value because of rounding. The rounding generally occurs on the sixth or seventh decimal digit. Also, when you cast down from `double` to `float`, you run the risk of causing a run-time overflow error caused by a `double` that is too big or too small to fit in a `float`.

Casting Pointers to Pointers

You may cast a pointer of one type to a pointer to any other type. For example:

```
int *int_p;
float *float_p;
struct S *str_p;
extern foo(struct T *);

int_p = (int *) float_p;
float_p = (float *) str_p;
foo((struct T *) str_p);
```

The cast is required whenever you assign a pointer value to a pointer variable that has a different base type, and when you pass a pointer value as a parameter to a function that has been prototyped with a different pointer type. The only exception to this rule concerns the generic pointer. You may assign any pointer value to a generic pointer without casting.

Comma Operator (,)

Syntax

exp1, exp2

Arguments

exp1

Any expression.

exp2

Any expression.

Description

Use the comma operator to separate two expressions that are to be evaluated one right after the other. The comma operator is popular within `for` loops, as demonstrated by the following example:

```
for (i = 10, j = 4; i * j < n; i++, j++);
```

In the preceding example, the comma operator allows you to initialize both `i` and `j` at the beginning of the loop. The comma operator also allows you to increment `i` and `j` together at the end of each loop iteration.

All expressions return values. When you use a comma operator, the expression returns the value of the rightmost expression. For example, the following statement sets variable `j` to 2:

```
j = (x = 1, y = 2);
```

Assignments such as these, however, are considered poor programming style. You should confine use of the comma operator to `for` loops.

Conditional Expression Operator (? :)

Syntax

exp1 ? *exp2* : *exp3*

Arguments

exp1

Any expression.

exp2

Any expression.

exp3

Any expression.

Description

The conditional expression construction provides a shorthand way of coding an `if...else` condition. The difference between the expression notation and an `if...else` condition is that the `? :` notation is an expression and therefore returns a value, while an `if...else` condition is a statement and does not return a value. The syntax described above is equivalent to

```
if (exp1)
    exp2;
else
    exp3;
```

When a conditional expression is executed, *exp1* is evaluated first. If it is true (that is, nonzero) *exp2* is evaluated and its result is the value of the conditional expression. If *exp1* is false, *exp3* is evaluated and its result is the value of the conditional expression.

There is no requirement that you put parentheses around the *exp1* portion of the conditional expression, but doing so will improve your code's readability.

Expressions and Operators

Conditional Expression Operator (? :)

Both *exp2* and *exp3* must be assignment-compatible. If *exp2* and *exp3* are pointers to different types, then the compiler issues a warning. The value of a conditional expression is either *exp2* or *exp3*, whichever is selected. The other expression is not evaluated. The type of the result is the type that would be produced if *exp2* and *exp3* were mixed in an expression. For instance, if *exp2* is a char and *exp3* is a double, the result type will be double regardless of whether *exp2* or *exp3* is selected.

Example

```
/* Program name is "conditional_exp_op_example".
   This program uses the conditional expression to
   see if the user wants to continue adding
   numbers. */
#include <stdio.h>

int main(void)
{
    int a, b, c, d, again, total;
    char answer;

    printf("\n");
    again = 1;
    while (again)
    {
        printf("Enter four numbers separated by spaces that\n");
        printf("you want added together: ");
        scanf("%d %d %d %d", &a, &b, &c, &d);
        fflush(stdin);
        total = a + b + c + d;
        printf("\nThe total is: %d\n", total);
        printf("Do you want to continue ? ");
        scanf("%c", &answer);
        again = (answer == 'y' || answer == 'Y') ? 1 : 0;
    } /* end while */
}
```

If you execute this program, you get the following output:

```
Enter four numbers separated by spaces that
you want added together: 20 30 40 50

The total is: 140
Do you want to continue ? y
Enter four numbers separated by spaces that
you want added together: 1 2 3 4

The total is: 10
Do you want to continue ? n
```

Function Calls

Syntax

postfix-expression ([*argument-expression-list*])

Description

Function calls provide a means of invoking a function and passing arguments to it.

The *postfix-expression* must have the type “pointer to function returning T”. The result of the function will be type T. Functions can return any type of object except array and function. Specifically, functions can return structures. In the case of structures, the contents of the returned structure is copied to storage in the calling function. For large structures, this can use a lot of execution time.

Although the expression denoting the called function must actually be a *pointer* to a function, in typical usage, it is simply a function name. This works because the function name will automatically be converted to a pointer.

C has no call statement. Instead, all function references must be followed by parentheses. The parentheses contain any arguments that are passed to the function. If there are no arguments, the parentheses must still remain. The parentheses can be thought of as a postfix *call operator*.

If the function name is not declared before it is used, the compiler enters the default declaration:

```
extern int identifier();
```

Function Arguments

Function arguments are expressions. Any type of object can be passed to a function as an argument. Specifically, structures can be passed as arguments. Structure arguments are copied to temporary storage in the called function. The length of time required to copy a structure argument depends upon the structure's size.

If the function being called has a prototype, each argument is evaluated and converted as if being assigned to an object of the type of the corresponding parameter. If the prototype has an ellipsis, any argument specified after the fixed parameters is subject to the *default argument promotions* described below.

The compiler checks to see that there are as many arguments as required by the function prototype. If the prototype has an ellipsis, additional parameters are allowed. Otherwise, they are flagged erroneous. Also, the types of the arguments must be assignment-compatible with their corresponding formal parameters, or the compiler will emit a diagnostic message.

If the function does not have a prototype, then the arguments are evaluated and subjected to the default argument promotions; that is, arguments of type `char` or `short` (both signed and unsigned) are promoted to type `int`, and `float` arguments are promoted to `double`.

In this case, the compiler does not do any checking between the argument types and the types of the parameters of the function (even if it has seen the definition of the function). Thus, for safety, it is highly advisable to use prototypes wherever possible.

In both cases, arrays of type `T` are converted to pointers to type `T`, and functions are converted to pointers to functions.

Function Formal Parameters

Within a function, the formal parameters are lvalues that can be changed during the function execution. This does not change the arguments as they exist in the calling function. It is possible to pass pointers to objects as arguments. The called function can then reference the objects indirectly through the pointers. The result is as if the objects were passed to the function using call by reference. The following `swap` function illustrates the use of pointers as arguments. The `swap()` function exchanges two integer values:

```
void swap(int *x,int *y)
{
    int t;

    t = *x;
    *x = *y;
    *y = t;
}
```

To swap the contents of integer variables `i` and `j`, you call the function as follows:

```
swap(&i, j);
```

Notice that the addresses of the objects (pointers to `int`) were passed and not the objects themselves. Because arrays of type `T` are converted into pointers to type `T`, you might think that arrays are passed to functions using *call by reference*. This is not actually the case. Instead, the address of the first element is passed to the called function. This is still strictly *call by value* since the pointer is passed by value. Inside the called function, references to the array via the passed starting address, are actually references to the array in the calling function. Arrays are not copied into the address space of the called function.

Function Recursion

All functions are recursive both in the direct and indirect sense. Function `A` can call itself directly or function `A` can call function `B` which, in turn, calls function `A`. Note that each invocation of a function requires program stack space. For this reason, the depth of recursion depends upon the size of the execution stack.

Increment and Decrement Operators (++, --)

Syntax

lvalue++

Increments the current value of *lvalue* after *lvalue* has been referenced.

lvalue--

Decrements the current value of *lvalue* after *lvalue* has been referenced.

++*lvalue*

Increments the current value of *lvalue* before *lvalue* is referenced.

--*lvalue*

Decrements the current value of *lvalue* after *lvalue* has been referenced.

Arguments

lvalue

Any previously declared integer or pointer *lvalue*. Although *lvalue* can be a pointer variable, it cannot be a pointer to a function.

Description

The increment operator (++) adds 1 to its operand. The decrement operator (--) subtracts 1 from its operand.

The increment and decrement operators are unary. The operand must be a scalar *lvalue* — it is illegal to increment or decrement a constant, structure, or union. It is legal to increment or decrement pointer variables, but the meaning of adding 1 to a pointer is different from adding 1 to an arithmetic value. This is described in “Pointer Operators (*, -, &)” on page 119.

Postfix and Prefix Forms

There are two forms for each of the operators: postfix and prefix. Both forms increment or decrement the appropriate variable, but they do so at different times. The statement `++i` (prefix form) increments `i` before using its value, while `i++` (postfix form) increments it after its value has been used. This difference can be important to your program.

The postfix increment and decrement operators fetch the current value of the variable and store a copy of it in a temporary location. The compiler then increments or decrements the variable. The temporary copy, which has the variable's value before it was modified, is used in the expression.

In many cases, you are interested only in the side effect, not in the result of the expression. In these instances, it doesn't matter whether you use postfix or prefix.

You need to be careful, however, when you use the increment and decrement operators within an expression.

Standalone Increment Decrement Expressions .

For example, as a stand-alone assignment or as the third expression in a for loop, the side effect is the same whether you use the prefix or postfix versions. The statement

```
x++;
```

is equivalent to

```
++x;
```

Similarly, the statement

```
for (j = 0; j <= 10; j++)
```

is equivalent to

```
for (j = 0; j <= 10; ++j)
```

Using Increment and Decrement within Expressions .

Consider the following function that inserts newlines into a text string at regular intervals.

```
#include <stdio.h>

void break_line(int interval)
{
    int c, j=0;
    while ((c = getchar()) != '\n') {
        if ((j++ % interval) == 0)
            printf("\n");
        putchar(c);
    }
}
```

This works because the postfix increment operator is used. If you use the prefix increment operator, the function breaks the first line one character early.

Side Effects of the Increment and Decrement Operators .

The increment and decrement operators and the assignment operators cause side effects. That is, they not only result in a value, but they change the value of a variable as well. A problem with side effect operators is that it is not always possible to predict the order in which the side effects occur. Consider the following statement:

```
x = j * j++;
```

The C language does not specify which multiplication operand is to be evaluated first. One compiler may evaluate the left operand first, while another evaluates the right operand first. The results are different in the two cases. If *j* equals 5, and the left operand is evaluated first, the expression will be interpreted as

```
x = 5 * 5; /* x is assigned 25 */
```

If the right operand is evaluated first, the expression becomes

```
x = 6 * 5; /* x is assigned 30 */
```

Statements such as this one are not portable and should be avoided. The side effect problem also crops up in function calls because the C language does not guarantee the order in which arguments are evaluated. For example, the function call

```
f(a, a++)
```

is not portable because compilers are free to evaluate the arguments in any order they choose.

To prevent side effect bugs, follow this rule: If you use a side effect operator in an expression, do not use the affected variable anywhere else in the expression. The ambiguous expression above, for instance, can be made unambiguous by breaking it into two assignments:

```
x = j * j;  
++j;
```

Precedence of Increment and Decrement Operators .

The increment and decrement operators have the same precedence, but bind from right to left. So the expression

```
--j++
```

is evaluated as

```
--(j++)
```

This expression is illegal because `j++` is not an lvalue as required by the operator. In general, you should avoid using multiple increment or decrement operators together.

Examples

```
i=k--; /* Stores the value of k in i then decrements k. */  
j=l++; /* Stores the value of l in j then increments l. */  
i--k; /* Decrements k then stores the new value of k in i. */  
j++l; /* Increments l then stores the new value of l in j. */
```

The following example uses both prefix and postfix increment and decrement operators:

```
#include <stdio.h>  
int main(void)  
{  
    int j = 5, k = 5, l = 5, m = 5;  
    printf("j: %d\t k: %d\n", j++, k--);  
    printf("j: %d\t k: %d\n", j, k);  
    printf("l: %d\t m: %d\n", ++l, --m);  
    printf("l: %d\t m: %d\n", l, m);  
}
```

The result is as follows:

```
j: 5    k: 5  
j: 6    k: 4  
l: 6    m: 4  
l: 6    m: 4
```

The results show that the *initial* values of `j` and `k` are used in the first `printf()`. They also show that `l` and `m` are incremented and decremented, respectively, *before* the third `printf()` call.

Logical Operators (&&, ||, !)

Syntax

exp1 && *exp2*

Logical AND.

exp1 || *exp2*

Logical OR.

!*exp1*

Logical NOT.

Arguments

exp1

Any expression.

exp2

Any expression.

Description

The logical AND operator (&&) and the logical OR (||) operator evaluate the truth or falsehood of pairs of expressions. The AND operator evaluates to 1 if and only if both expressions are true. The OR operator evaluates to 1 if *either* expression is true. To test whether *y* is greater than *x* and less than *z*, you would write

```
(x < y) && (y < z)
```

The logical negation operator (!) takes only one operand. If the operand is true, the result is false; if the operand is false, the result is true.

The operands to the logical operators may be integers or floating-point objects. The expression

```
1 && -5
```

results in 1 because both operands are nonzero. The same is true of the expression

0.5 && -5

Logical operators (and the comma and conditional operators) are the only operators for which the order of evaluation of the operands is defined. The compiler must evaluate operands from left to right. Moreover, the compiler is guaranteed not to evaluate an operand if it is unnecessary. For example, in the expression

```
if ((a != 0) && (b/a == 6.0))
```

if *a* equals 0, the expression (*b/a* == 6) will not be evaluated. This rule can have unexpected consequences when one of the expressions contains side effects.

Truth Table for C's Logical Operators

In C, true is equivalent to any *nonzero* value, and false is equivalent to 0. The following table shows the logical tables for each operator, along with the numerical equivalent. All of the operators return 1 for true and 0 for false.

Table 5-10

Truth Table for C's Logical Operators

Operand	Operator	Operand	Result
zero	&&	zero	0
nonzero	&&	zero	0
zero	&&	nonzero	0
nonzero	&&	nonzero	1
zero		zero	0
nonzero		zero	1
zero		nonzero	1
nonzero		nonzero	1
not applicable	!	zero	1
	!	nonzero	0

Examples of Expressions Using the Logical Operators

The following table shows a number of examples that use relational and logical operators. The logical NOT operator has a higher precedence than the others. The AND operator has higher precedence than the OR operator. Both the logical AND and OR operators have lower precedence than the relational and arithmetic operators.

Table 5-11 **Examples of Expressions Using the Logical Operators**

Given the following declarations:		
<pre>int j = 0, m = 1, n = -1; float x = 2.5, y = 0.0;</pre>		
Expression	Equivalent Expression	Result
<code>j && m</code>	<code>(j) && (m)</code>	0
<code>j < m && n < m</code>	<code>(j < m) && (n < m)</code>	1
<code>m + n !j</code>	<code>(m + n) (!j)</code>	1
<code>x * 5 && 5 m / n</code>	<code>((x * 5) && 5) (m / n)</code>	1
<code>j <= 10 && x >= 1 && m</code>	<code>((j <= 10) && (x >= 1)) && m</code>	1
<code>!x !n m+n</code>	<code>((!x) (!n)) (m+n)</code>	0
<code>x * y < j + m n</code>	<code>((x * y) < (j + m)) n</code>	1
<code>(x > y) + !j n++</code>	<code>((x > y) + (!j)) (n++)</code>	1
<code>(j m) + (x ++n)</code>	<code>(j m) + (x (++n))</code>	2

Side Effects in Logical Expressions

Logical operators (and the conditional and comma operators) are the only operators for which the order of evaluation of the operands is defined. For these operators, operands must be evaluated from left to right. However, the system evaluates only as much of a logical expression as it needs to determine the result. In many cases, this means that the system does not need to evaluate the entire expression. For instance, consider the following expression:

```
if ((a < b) && (c == d))
```

The system begins by evaluating $(a < b)$. If a is not less than b , the system knows that the entire expression is false, so it will not evaluate $(c == d)$. This can cause problems if some of the expressions contain side effects:

```
if ((a < b) && (c == d++))
```

In this case, d is only incremented when a is less than b . This may or may not be what the programmer intended. In general, you should avoid using side effect operators in logical expressions.

Example

```
/* Program name is "logical_ops_example". This program */
/* shows how logical operators are used. */
#include <stdio.h>

int main(void)
{
    int won_lottery, enough_vacation, money_saved;
    char answer;

    won_lottery = enough_vacation = money_saved = 0;

    printf("\nThis program determines whether you can ");
    printf("take your next vacation in Europe.\n");
    printf("Have you won the lottery? y or n: ");
    fflush(stdin);
    scanf("%c", &answer);
    if (answer == 'y')
        won_lottery = 1;

    printf("Do you have enough vacation days saved? \
y or n: ");
    fflush(stdin);
    scanf("%c", &answer);
    if (answer == 'y')
        enough_vacation = 1;

    printf("Have you saved enough money for the trip? \
y or n: ");
    fflush(stdin);
    scanf("%c", &answer);
    if (answer == 'y')
        money_saved = 1;

    printf("\n");
    if (won_lottery)
    {
        printf("Why do you need a program to decide if you");
        printf(" can afford a trip to Europe?\n");
    } /* end if */
    if (won_lottery || (enough_vacation && money_saved))
        printf("Look out Paris!\n");
    else if (enough_vacation && (!money_saved))
        printf("You've got the time, but you haven't got \
```

Expressions and Operators

Logical Operators (&&, ||, !)

```
the dollars.\n");
    else if (!enough_vacation || (!money_saved))
    {
        printf("Tough luck. Try saving your money and ");
        printf("vacation days next year.\n");
    } /* end else/if */
}
```

If you execute this program, you get the following output:

```
This program determines whether you can take your next vacation
in Europe.
Have you won the lottery? y or n: y
Do you have enough vacation days saved? y or n: n
Have you saved enough money for the trip? y or n: n

Why do you need a program to decide if you can afford a trip to
Europe?
Look out Paris!
```


Pointer Operators (*, ->, &)

Syntax

**ptr_exp*

Dereferences a pointer. That is, finds the contents stored at the virtual address that *ptr_exp* holds.

ptr->member

Dereferences a *ptr* to a structure or union where *member* is a member of that structure or union.

&lvalue

Finds the virtual address where the lvalue stored.

Description

A pointer variable is a variable that can hold the address of an object.

Assigning an Address Value to a Pointer

To declare a pointer variable, you precede the variable name with an asterisk. The following declaration, for example, makes `ptr` a variable that can hold addresses of `long int` variables:

```
long *ptr;
```

The data type, `long` in this case, refers to the type of variable that `ptr` can point to. To assign a pointer variable with the virtual address of a variable, you can use the address-of operator `&`. For instance, the following is legal:

```
long *ptr;
long long_var;
ptr = &long_var; /* Assign the address of long_var to ptr. */
```

But this is illegal:

```
long *ptr;
float float_var;
ptr = &float_var; /* ILLEGAL - because ptr can only store the
                  address of a long int. */
```

The following program illustrates the difference between a pointer variable and an integer variable.

Expressions and Operators

Pointer Operators (*, -, &)

```
/* Program name is "ptr_example1". */
#include <stdio.h>
int main()
{
    int j = 1;
    int *pj;
    pj = &j; /* Assign the address of j to pj */
    printf("The value of j is: %d\n", j);
    printf("The address of j is: %p\n", pj);
}
```

If you run this program (in 32-bit mode), the output looks something like this:

```
The value of j is: 1
The address of j is: 7b033240
```

Dereferencing a Pointer

To dereference a pointer (get the value stored at the pointer address), use the * operator. The program below shows how dereferencing works:

```
/* Program name is "ptr_example2". */
#include <stdio.h>

int main(void)
{
    char *p_ch;
    char ch1 = 'A', ch2;
    printf("The address of p_ch is %p\n", &p_ch);
    p_ch = &ch1;
    printf("The value of p_ch is %p\n", p_ch);
    printf("The dereferenced value of p_ch is %c\n",
           *p_ch);
}
```

The output from this program looks something like this:

```
The address of p_ch is 7b033240
The value of p_ch is 7b033244
The dereferenced value of p_ch is A
```

This is a roundabout and somewhat contrived example that assigns the character A to both ch1 and ch2. It does, however, illustrate the effect of the dereference (*) operator. The variable ch1 is initialized to A. The first printf() call displays the address of the pointer variable p_ch. In the next step, p_ch is assigned the address of ch1, which is also displayed. Finally, the dereferenced value of p_ch is displayed and ch2 is assigned to it.

The expression `*p_ch` is interpreted as “Take the address value stored in `p_ch` and get the value stored at that address.” This gives us a new way to look at the declaration. The data type in the pointer declaration indicates what type of value results when the pointer is dereferenced. For instance, the declaration

```
float *fp;
```

means that when `*fp` appears as an expression, the result will be a float value.

The expression `*fp` can also appear on the left side of an expression:

```
*fp = 3.15;
```

In this case, we are storing a value (3.15) at the location designated by the pointer `fp`. This is different from

```
fp = 3.15;
```

which attempts to store the address 3.15 in `fp`. This, by the way, is illegal, because addresses are not the same as floating-point values.

When you assign a value through a dereferenced pointer, make sure that the data types agree. For example:

```
/* Program name is "ptr_example3". */
#include <stdio.h>

int main(void)
{
    float f = 1.17e3, g;
    int *ip;
    ip = &f;
    g = *ip;
    printf("The value of f is: %f\n", f);
    printf("The value of g is %f\n", g);
}
```

The result is

```
The value of f is: 1170.000000
The value of g is: 1150435328.000000
```

In the preceding example, instead of getting the value of `f`, `g` gets an erroneous value because `ip` is a pointer to an `int`, not a float. The HP C compiler issues a warning message when a pointer type is unmatched. If you compile the preceding program, for instance, you receive the following message:

```
cc: "ptr_example3.c", line 9: warning 604: Pointers are not
assignment-compatible.
```

Pointer Arithmetic

The following arithmetic operations with pointers are legal:

You may add an integer to a pointer or subtract an integer from a pointer.

You may use a pointer as an operand to the ++ and -- operators.

You may subtract one pointer from another pointer, if they point to objects of the same type.

You may compare two pointers

All other arithmetic operations with pointers are illegal.

When you add or subtract an integer to or from a pointer, the compiler automatically scales the integer to the pointer's type. In this way, the integer always represents the number of objects to jump, not the number of bytes. For example, consider the following program fragment:

```
int x[10], *p1x = x, *p2x;  
p2x = p1x + 3;
```

Since pointer `p1x` points to a variable (`x`) that is 4 bytes long, then the expression `p1x + 3` actually increments `p1x` by 12 ($4 * 3$), rather than by 3.

It is legal to subtract one pointer value from another, provided that the pointers point to the same type of object. This operation yields an integer value that represents the number of objects between the two pointers. If the first pointer represents a lower address than the second pointer, the result is negative. For example,

```
&a[3] - &a[0]  
evaluates to 3, but
```

```
&a[0] - &a[3]  
evaluates to -3.
```

It is also legal to subtract an integral value from a pointer value. This type of expression yields a pointer value. The following examples illustrate some legal and illegal pointer expressions:

```
long *p1, *p2;  
int a[5], j;  
char *p3;  
  
p1 = a;      /* Same as p1 = &a[0] */  
p2 = p1 + 4; /* legal */
```

```
j = p2 - p1; /* legal -- j is assigned 4 */
j = p1 - p2; /* legal -- j is assigned -4 */
p1 = p2 - 2; /* legal -- p2 points to a[2] */
p3 = p1 - 1; /* ILLEGAL -- different pointer types*/
j = p1 - p3; /* ILLEGAL -- different pointer types*/
j = p1 + p2; /* ILLEGAL -- can't add pointers */
```

Arrays and Pointers

Arrays and pointers have a close relationship in the C language. You can exploit this relationship in order to write more efficient code. See the discussion of “Array Subscripting ([])” on page 84 for more information.

Casting a Pointer's Type

A pointer to one type may be cast to a pointer to any other type. For example, in the following statements, a pointer to an `int` is cast to a pointer to a `char`. Presumably, the function `func()` expects a pointer to a `char`, not a pointer to an `int`.

```
int i, *p = &i;
func((char *) p);
```

As a second example, a pointer to a `char` is cast to a pointer to `struct H`:

```
struct H {
    int q;
} x, y;
char *genp = &x;
y = (struct H *)genp->q;
```

See “Cast Operator” on page 99 for more information about the cast operator.

It is always legal to assign any pointer type to a generic pointer, and vice versa, without a cast. For example:

```
float x, *fp = &x;
int j, *pj = &j;
void *pv;
pv = fp; /* legal */
fp = pv; /* legal */
```

In both these cases, the pointers are implicitly cast to the target type before being assigned.

Null Pointers

The C language supports the notion of a **null pointer** — that is, a pointer that is guaranteed not to point to a valid object. A null pointer is any pointer assigned the integral value 0. For example:

Expressions and Operators

Pointer Operators (*, ->, &)

```
char *p;
p = 0; /* make p a null pointer */
```

In this one case — assignment of 0 — you do not need to cast the integral expression to the pointer type.

Null pointers are particularly useful in control-flow statements, since the zero-valued pointer evaluates to false, whereas all other pointer values evaluate to true. For example, the following `while` loop continues iterating until `p` is a null pointer:

```
char *p;
while (p) {
    /* iterate until p is a null pointer */
    . . .
}
```

This use of null pointers is particularly prevalent in applications that use arrays of pointers.

The compiler does not prevent you from attempting to dereference a null pointer; however, doing so may trigger a run-time access violation. Therefore, if it is possible that a pointer variable is a null pointer, you should make some sort of test like the following when dereferencing it:

```
if (px && *px) /* if px = 0, expression will short-circuit
    . . .      before dereferencing occurs*/
```

Null pointers are a portable feature.

Example 1

```
/* Program name is "pointer_array_example1". This program
 * shows how to access a 1-dimensional array through
 * pointers. Function count_chars returns the number of
 * characters in the string passed to it.
 * Note that *arg is equivalent to a_word[0];
 * arg + 1 is equivalent to a_word[1]...
 */
#include <stdio.h>

int count_chars(char *arg)
{
    int count = 0;
    while (*arg++)
        count++;
    return count;
}

int main(void)
{
    char a_word[30];
    int number_of_characters;
```

```
printf("Enter a word -- ");
scanf("%s", a_word);
number_of_characters = count_chars(a_word);
printf("%s contains %d characters.\n", a_word,
       number_of_characters);
}
```

If you execute this program, you get the following output:

```
Enter a word -- Marilyn
Marilyn contains 7 characters.
```

Example 2

```
/* Program name is "pointer_array_example2". This program
 * demonstrates two ways to access a 2-dimensional array.
 */
#include <stdio.h>

int main(void)
{
    int count = 0, which_name;
    char c1, c2;
    static char str[5][10] = {"Phil", "Sandi", "Barry",
                             "David", "Amy"};
    static char *pstr[5] = { str[0], str[1], str[2],
                             str[3], str[4]};
    /* pstr is an array of pointers. Each element in the array
     * points to the beginning of one of the arrays in str.
     */
    /* Prompt for information. */
    printf("Which name do you want to retrieve?\n");
    printf("Enter 0 for the first name,\n");
    printf("      1 for the second name, etc. -- ");
    scanf("%d", &which_name);
    /* Print name directly through array. */
    while (c1 = str[which_name][count++])
        printf("%c", c1);
    printf("\n");

    /* Print same name indirectly through an array of pointers. */
    while (c2 = *(pstr[which_name]++))
        printf("%c", c2);
    /* We could also have used the following statement instead of
     * the two previous ones: printf("%s", pstr[which_name]);
     */
    printf("\n");
}
```

Expressions and Operators
Pointer Operators (*, ->, &)

If you execute this program, you get the following output:

```
Which name do you want to retrieve?  
Enter 0 for the first name,  
      1 for the second name, etc. -- 1  
Sandi  
Sandi
```

Relational Operators (>, >=, <, ==, !=)

Syntax

exp1 > exp2

Greater than.

exp1 >= exp2

Greater than or equal to.

exp1 < exp2

Less than.

exp1 <= exp2

Less than or equal to.

exp1 == exp2

Equal to.

exp1 != exp2

Not equal to.

Arguments

exp1

Any expression.

exp2

Any expression.

Description

A relational expression consists of two expressions separated by one of six relational operators. The relational expression evaluates either to 1 (true) or 0 (false).

The equality operator (==) performs the same function as Pascal's = or Fortran's .EQ.; it just looks different. Although the equality operator looks similar to the assignment operator (=), the two operators serve

completely different purposes. Use the assignment operator when you want to assign a value to a variable, but use the equality operator when you want to test the value of an expression.

Confusing = with ==

One of the most common mistakes made by beginners and experts alike is to use the assignment operator (=) instead of the equality operator (==). For instance:

```
while (j = 5)
    do_something();
```

What is intended, clearly, is that the `do_something()` function should only be invoked if `j` equals five. It should be written

```
while (j == 5)
    do_something();
```

The first version is syntactically legal, since all expressions have a value. The value of the expression `j = 5` is 5. Since this is a nonzero value, the while expression will always evaluate to true and `do_something()` will always be invoked.

Relational Operators Precedence Rules

Relational operators have lower precedence than arithmetic operators. The expression

```
a + b * c < d / f
```

is evaluated as if it had been written

```
(a + (b * c)) < (d / f)
```

Among the relational operators, `>`, `>=`, `<`, and `<=` have the same precedence. The `==` and `!=` operators have lower precedence. All of the relational operators have left-to-right associativity. The following table illustrates how the compiler parses complex relational expressions.

Table 5-12 **Examples of Expressions Using the Relational Operators**

Given the following declaration:		
<pre>int j = 0, m = 1, n = -1; float x = 2.5, y = 0.0;</pre>		
Expression	Equivalent Expressions	Result
<code>j > m</code>	<code>j > m</code>	0
<code>m / n < x</code>	<code>(m / n) < x</code>	1
<code>j <= m >= n</code>	<code>((j <=m) >= n)</code>	1
<code>j <= x == m</code>	<code>((j <= x) == m)</code>	1
<code>- x + j == y > n > m</code>	<code>((-x) + j) == ((y > n) >= m)</code>	0
<code>x += (y >= n)</code>	<code>x = (x + (y >= n))</code>	3.5
<code>++j == m != y * 2</code>	<code>((++j) == m) != (y * 2)</code>	1

Evaluation of Relational Expressions

Relational expressions are often called Boolean expressions, in recognition of the nineteenth-century mathematician and logician, George Boole. Many programming languages, such as Pascal, have Boolean data types for representing true and false. The C language, however, represents these values with integers. Zero is equivalent to false, and any nonzero value is considered true.

The value of a relational expression is an integer, either 1 (indicating the expression is true) or 0 (indicating the expression is false). The examples in the following table illustrate how relational expressions are evaluated:

Table 5-13 **Relational Expressions**

Expression	Value
<code>-1 < 0</code>	1
<code>0 > 1</code>	0
<code>5 == 5</code>	1

Expressions and Operators

Relational Operators (>, >=, <, ==, !=)

Expression	Value
7 != -3	1
1 >= -1	1
1 > 10	0

Because Boolean values are represented as integers, you can write

```
if (j)
    statement;
```

If *j* is any nonzero value, *statement* is executed; if *j* equals 0, *statement* is skipped. Likewise, the statement

```
if (isalpha(ch))
```

is exactly the same as

```
if (isalpha(ch) != 0)
```

The practice of using a function call as a Boolean expression is a common idiom in C. It is especially effective for functions that return 0 if an error occurs, since you can use a construct such as

```
if (func())
    proceed;
else
    error_handler;
```

Dangers of Comparing Floating-Point Values

You may get unexpected results if you compare floating-point values for equality because floating-point representations are inexact for some numbers. For example, the following expression, though algebraically true, will evaluate to false on many computers:

```
(1.0/3.0 + 1.0/3.0 + 1.0/3.0) == 1.0
```

This evaluates to 0 (false) because the fraction 1.0/3.0 contains an infinite number of decimal places (3.33333...). The computer is only capable of holding a limited number of decimal places, so it rounds each occurrence of 1/3. As a result, the left side of the expression does not equal exactly 1.0.

This problem can occur in even more subtle ways. Consider the following code:

```
double divide(double num, double denom)
{
    return num/denom;
}
int main(void)
{
    double c, a = 1.0, b = 3.0;
    c = a/b;
    if (c != divide(a, b))
        printf("Fuzzy doubles\n");
}
```

Surprisingly, the value stored in `c` may not equal the value returned by `divide()`. This anomaly occurs due to the fact that some computers can represent more decimal places for values stored in registers than for values stored in memory. Because the value returned by `divide()` is never stored in memory, it may not be equal to the value `c`, which has been rounded for memory storage.

To avoid bugs caused by inexact floating-point representations, you should refrain from using strict equality comparisons with floating-point types.

Expressions and Operators

Relational Operators (>, >=, <, ==, !=)

Example

```
/* Program name is "relational_example". This program
 * does some mathematical calculations and shows
 * C's relational operators in action.
 */
#include <stdio.h>

int main(void)
{
    int num, i;
    printf("\n");
    num = 5;
    printf("The number is: %d\n", num);
    for (i = 0; i <= 2; i++)
    {
        if (num < 25)
        {
            num *= num;
            printf("The number squared is: %d\n", num);
        }
        else if (num == 25) {
            num *= 2;
            printf("Then, when you double that, you get: %d\n",
num);
        }
        else if (num > 25)
        {
            num -= 45;
            printf("And when you subtract 45, you're back where ");
            printf("you started at: %d\n", num);
        } /* end if */
    } /* end for */

    if (num != 5)
        printf("The programmer made an error in setting up this \
example\n");
}
```

If you execute this program, you get the following output:

```
The number is: 5
The number squared is: 25
Then, when you double that, you get: 50
And when you subtract 45, you're back where you started at: 5
```

sizeof Operator

Syntax

```
sizeof exp;  
sizeof (type_name)
```

Arguments

exp

An expression of any type except function, void, or bit field.

type_name

The name of a predefined or user-defined data type, or the name of some variable. An example of a predefined data type is `int`. A user-defined data type could be the tag name of a structure.

Description

The `sizeof` unary operator finds the size of an object. The `sizeof` operator accepts two types of operands: an expression or a data type. If you use an expression operand, the expression itself is not evaluated — the compiler only determines what type the result would be. Any side effects in the expression, therefore, will not have an effect. The result type of the `sizeof` operator is `unsigned int`.

If the operand is an expression, `sizeof` returns the number of bytes that the result occupies in memory:

```
/* Returns the size of an int (4 if ints are four  
 * bytes long)  
 */  
sizeof(3 + 5)  
  
/* Returns the size of a double (8 if doubles are  
 * eight bytes long)  
 */  
sizeof(3.0 + 5)  
  
/* Returns the size of a float (4 if floats are  
 * four bytes long)  
 */  
float x;  
sizeof(x)
```

Expressions and Operators

sizeof Operator

For expressions, the parentheses are optional, so the following is legal:

```
sizeof x
```

By convention, however, the parentheses are usually included.

The operand can also be a data type, in which case the result is the length in bytes of objects of that type:

```
sizeof(char)    /* 1 on all machines */
sizeof(short)   /* 2 on HP 9000 Series */
sizeof(float)   /* 4 on HP 9000 Series */
sizeof(int *)   /* 4 on HP 9000 Series */
```

The parentheses are required if the operand is a data type.

The results of most `sizeof` expressions are implementation dependent. The only result that is guaranteed is the size of a `char`, which is always 1.

In general, the `sizeof` operator is used to find the size of aggregate data objects such as arrays and structures.

Example

You can use the `sizeof` operator to obtain information about the sizes of objects in your C environment. The following prints the sizes of the basic data types:

```
/* Program name is "sizeof_example". This program
 * demonstrates a few uses of the sizeof operator.
 */
#include <stdio.h>
int main(void)
{
    printf("TYPE\t\tSIZE\n\n");
    printf("char\t\t%d\n", sizeof(char));
    printf("short\t\t%d\n", sizeof(short));
    printf("int\t\t%d\n", sizeof(int));
    printf("float\t\t%d\n", sizeof(float));
    printf("double\t\t%d\n", sizeof(double));
}
```

If you execute this program, you get the following output:

TYPE	SIZE
char	1
short	2
int	4
float	4
double	8

Structure and Union Members (., ->)

A member of a structure or a union can be referenced using either of two operators: the period or the right arrow.

Syntax

postfix-expression . identifier postfix-expression -> identifier

Description

Use the period to reference members of structures and unions directly. Use the arrow operator to reference members of structures and unions pointed to by pointers. The arrow operator combines the functions of indirection through a pointer and member selection. If P is a pointer to a structure with a member M , the expression $P \rightarrow M$ is identical to $(*P) . M$.

The *postfix-expression* in the first alternative must be a structure or a union. The expression is followed by a period (.) and an identifier. The identifier must name a member defined as part of the structure or union referenced in the *postfix-expression*. The value of the expression is the value of the named member. It is an lvalue if the *postfix-expression* is an lvalue.

If the *postfix-expression* is a pointer to a structure or a pointer to a union, follow it with an arrow (composed of the - character followed by the |) and an identifier. The identifier must name a member of the structure or union which the pointer references. The value of the primary expression is the value of the named member. The resulting expression is an lvalue.

The . operator and the -> operator are closely related. If S is a structure, M is a member of structure S , and $\&S$ is a valid pointer expression, $S . M$ is the same as $(\&S) \rightarrow M$.

Operator Precedence

Precedence is the order in which the compiler groups operands with operators. The C compiler evaluates certain operators and their operands before others. If operands are not grouped using parentheses, the compiler groups them according to its own rules.

The following lists C operator precedence in *highest to lowest* precedence:

Table 5-14

C Operator Precedence

Class of operator	Operators	Grouping
primary	() [] -> .	left to right
unary	(type casting) sizeof & (address of) * (dereference) - (reverse sign) ~ ! ++ --	right to left
multiplicative	* / %	left to right
additive	+ -	left to right
shift	<< >>	left to right
relational	< <= > >=	left to right
equality	== !=	left to right
bitwise AND	&	left to right
bitwise XOR	^	left to right
bitwise OR		left to right
logical AND	&&	left to right
logical OR		left to right

Class of operator	Operators	Grouping
conditional	? :	right to left
assignment	= += -= *= /= %= >>= <<= &= ^= =	right to left
comma	,	left to right

Precedence among Operators of Same Class

Most operators group from the left to the right but some group from the right to the left. The grouping indicates how an expression containing several operators of the same precedence will be evaluated. Left to right grouping means the expression

$a/b * c/d$

behaves as if it had been written:

$((a/b)*c)/d$

Likewise, an operator that groups from the right to the left causes the expression

$a = b = c$

to behave as if it had been written:

$a = (b = c)$

Operator Quick Reference

Table 5-15

C Operators

Symbol	Meaning
!	logical negation
!=	inequality
%	remainder
&	AND (bitwise) and address-of
&&	AND (logical)
()	cast and function call
*	multiplication and indirection
+	addition and unary plus
++	increment, prefix or postfix
,	comma
-	subtraction and unary minus
--	decrement, prefix or postfix
->	structure/union pointer (arrow)
.	structure/union member (dot)
/	division
<	less-than
<<	left-shift
<=	less-than-or-equal-to
=	assignment
==	equality
>	greater-than

Symbol	Meaning
>=	greater-than-or-equal-to
>>	right shift
?:	conditional
[]	subscript
^	OR (bitwise exclusive)
	OR (bitwise inclusive)
	OR (logical)
~	complement
<i>op</i> =	assignment, compound
sizeof	compute object size at translation-time

Constant Expressions

Constant expressions contain only constant values. For example, the following are all constant expressions:

```
5  
5 + 6 * 13 / 3.0  
'a'
```

Integral Expressions

Integer expressions are expressions that, after all automatic and explicit type conversions, produce a result that has one of the integer types. If `j` and `k` are integers, the following are all integral expressions:

```
j
j * k
j / k + 3
k - 'a'
3 + (int) 5.0
```

Floating-Point Expressions

Floating-point expressions are expressions that, after all automatic and explicit type conversions, produce a result that has one of the floating-point types. If `x` is a `float` or `double`, the following are floating-point expressions:

```
x
x + 3
x / y * 5
3.0
3.0 - 2
3 + (float) 4
```

lvalue Expressions

An **lvalue** (pronounced “el-value”) is an expression that refers to a region of storage that can be manipulated.

For example, all simple variables, like `ints` and `floats` are lvalues. An element of an array is also an lvalue; however an entire array is not. A member of a structure or union is an lvalue; an entire structure or union is not.

Given the following declarations:

```
int *p, a, b;
int arr[4];
int func();

a           /* lvalue */
a + b       /* Not an lvalue */
p           /* lvalue */
*p          /* lvalue */
arr         /* lvalue, but not modifiable */
*(arr + a)  /* lvalue */
arr[a]      /* lvalue, equivalent to *(arr+a) */
func        /* Not an lvalue */
func()      /* Not an lvalue */
```

Pointer Expressions

Pointer expressions are expressions that evaluate to an address value. These include expressions containing pointer variables, the address-of operator (`&`), string literals, and array names. If `p` is a pointer and `j` is an `int`, the following are pointer expressions:

```
p
&j
p + 1
"abc"
(char *) 0x000fffff
```

Evaluation of Expressions

Expressions are evaluated at run time. The results of the evaluation are called byproduct values. For many expressions, you won't know or care what this byproduct is. In some expressions, though, you can exploit this feature to write more compact code.

Examples

The following expression is an assignment.

```
x = 6;
```

The value 6 is both the byproduct value and the value that gets assigned to `x`. The byproduct value is not used.

The following example uses the byproduct value:

```
y = x = 6;
```

The equals operator binds from right to left; therefore, C first evaluates the expression `x = 6`. The byproduct of this operation is 6, so C sees the second operation as

```
y = 6
```

Now, consider the following relational operator expression:

```
(10 < j < 20)
```

It is incorrect to use an expression like this to find out whether `j` is between 10 and 20. Since the relational operators bind from left to right, C first evaluates

```
10 < j
```

The byproduct of a relational operation is 0 if the comparison is false and 1 if the comparison is true.

Assuming that `j` equals 5, the expression `10 < j` is false. The byproduct will be 0. Thus, the next expression evaluated:

```
0 < 20
```

is true (or 1). This is not the expected answer when `j` equals 5.

Finally, consider the following fragment:


```
static char a_char, c[20] = {"Valerie"}, *pc = c;
while (a_char = *pc++) {
    .
    .
}
```

This `while` statement uses C's ability to both assign and test a value. Every iteration of `while` assigns a new value to variable `a_char`. The byproduct of an assignment is equal to the value that gets assigned. The byproduct value will remain nonzero until the end of the string is reached. When that happens, the byproduct value will become 0 (false), and the `while` loop will end.

Evaluation Order of Subexpressions

The C language does not define the evaluation order of subexpressions within a larger expression except in the special cases of the `&&`, `||`, `?:`, and `,` operators. When programming in other computer languages, this may not be a concern. C's rich operator set, however, introduces operations that produce side effects. The `++` operator is a prime example. The `++` operator increments a value by 1 and provides the value for further calculations. For this reason, expressions such as

```
b = ++a*2 + ++a*4;
```

are dangerous. The language does not specify whether the variable `a` is first incremented and multiplied by 4 or is first incremented and multiplied by 2. The value of this expression is undefined.

Expressions and Operators
Evaluation of Expressions

6

Statements

Statements are the executable parts of a C function. All statements end with a semicolon. Statements are grouped as follows:

- Branch Statements
- Compound Statement or Block
- Expression and Null Statements
- Labeled Statements
- Looping Statements
- Selection Statements

If you know the name of the statement for which you need help, you can go to this section:

- break
 - continue
 - do...while
 - for
 - goto
 - if
 - return
 - switch
 - while
-

Branch Statements

Syntax

```
goto label;  
goto *expression;  
continue;  
break;  
return [expression];
```

Description

Branch statements transfer control unconditionally to another place in the executing program. The branch statements are `goto`, `continue`, `break`, and `return`.

Examples

These four fragments all accomplish the same thing (they print out the multiples of 5 between 1 and 100):

```
i = 0;  
while (i < 100)  
{  
    if (++i % 5)  
        continue; /* unconditional jump to top of while loop */  
    printf ("%2d ", i);  
}  
printf ("\n");  
  
i = 0;  
L: while (i < 100)  
{  
    if (++i % 5)  
        goto L; /* unconditional jump to top of while loop */  
    printf ("%2d ", i);  
}  
printf ("\n");  
  
i = 0;  
while (1)  
{  
    if ((++i % 5) == 0)  
        printf ("%2d ", i);  
    if (i > 100)  
        break; /* unconditional jump past the while loop */  
}  
printf ("\n");
```

```
i = 0;
while (1)
{
    if ((++i % 5) == 0)
        printf ("%2d ", i);
    if (i > 100) {
        printf ("\n");
        return; /* unconditional jump to calling function */
    }
}
```

Compound Statement or Block

Syntax

compound-statement ::=
{ [*declaration-list*] [*statement-list*] }

declaration-list ::=
 declaration
 declaration-list declaration

statement-list ::=
 statement
 statement-list statement

Description

A **compound statement** allows you to group statements together in a block of code and use them as if they were a single statement.

Variables and constants declared in the block are local to the block and to any inner blocks unless declared `extern`. If the objects are initialized, the initialization is performed each time the compound statement is entered from the top through the left brace (`{`) character. If the statement is entered via a `goto` statement or in a `switch` statement, the initialization is not performed.

Any object declared with static storage duration is created and initialized when the program is loaded for execution. This is true even if the object is declared in an inner block.

Example

```
if (x > y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

In this example, variable `temp` is local to the compound statement. It can only be accessed within the compound statement.

Expression and Null Statements

Syntax

expression-statement ::=
[*expression*] ;

Description

C **expressions** can be statements. A **null statement** is simply a semicolon by itself.

You can use any valid expression as an expression statement by terminating it with a semicolon. Expression statements are evaluated for their side effects such as assignment or function calls. If the expression is not specified, but the semicolon is still provided, the statement is treated as a null statement.

Null statements are useful for specifying no-operation statements. No-operation statements are often used in looping constructs where all of the work of the statement is done without an additional statement.

Example

A program fragment that sums up the contents of an array named `x` containing 10 integers might look like this:

```
for(i=0,s=0; i<10; s+=x[i++]);
```

The syntax of the `for` statement requires a statement following the closing `)` of the `for`. A null statement (`;`) satisfies this syntax requirement.

Labeled Statements

Syntax

```
labeled-statement ::=  
identifier : statement  
case constant-expression : statement  
default: statement
```

Description

Labeled statements are those preceded by a name or tag. You can prefix any statement using a label so at some point you can reference it using `goto` statements. Any statement can have one or more labels.

The `case` and `default` labels can only be used inside a `switch` statement.

Example

```
if (fatal_error)  
    goto get_out;  
    .  
    .  
get_out: return(FATAL_CONDITION);
```

The `return` statement is labeled `get_out`.

Looping Statements

Syntax

```
while (expression) statement  
do statement while (expression);  
for ([expression1] ; [expression2] ; [expression3])  
    statement
```

Description

You use looping statements to force a program to repeatedly execute a statement. The executed statement is called the *loop body*. Loops execute until the value of a controlling expression is 0. The controlling expression may be any scalar data type.

C has several looping statements: `while`, `do...while`, and `for`. The main difference between these statements is the point at which each loop tests for the exit condition. Refer to the `goto`, `continue`, and `break` statements for ways to exit a loop without reaching its end or meeting loop exit tests.

Examples

These three loops all accomplish the same thing (they assign `i` to `a[i]` for `i` from 0 to 4):

```
i = 0;  
while (i < 5)  
{  
    a[i] = i;  
    i++;  
}  
  
i = 0;  
do  
{  
    a[i] = i;  
    i++;  
} while (i < 5);  
  
for (i = 0; i < 5; i++)  
{  
    a[i] = i;  
}
```

Selection Statements

Syntax

```
if (expression) statement [else statement]  
switch (expression) statement
```

Description

A selection statement alters a program's execution flow by selecting one path from a collection based on a specified controlling expression. The `if` statement and the `switch` statement are selection statements.

Examples

```
if (expression) statement:  
    if (x<y) x=y;  
  
if (expression) statement else statement:  
    if (x<y) x=y; else y=x;  
  
switch (expression) statement:  
    switch (x)  
    { case 1: x=y;  
      break;  
      default: y=x;  
        break;  
    }
```

break

Syntax

```
break;
```

Description

A `break` statement terminates the execution of the most tightly enclosing `switch` statement or `for`, `while`, `do...while` loop.

Control passes to the statement following the `switch` or iteration statement. You cannot use a `break` statement unless it is enclosed in a `switch` or loop statement. Further, a `break` only exits out of one level of `switch` or loop statement. To exit from more than one level, you must use a `goto` statement.

When used in the `switch` statement, `break` normally terminates each case statement. If you use no `break` (or other unconditional transfer of control), each statement labeled with `case` flows into the next. Although not required, a `break` is usually placed at the end of the last case statement. This reduces the possibility of errors when inserting additional cases at a later time.

Example

The following example uses `break` to exit from the `for` loop after executing the loop three times:

```
for (i=0; i<=6; i++)
    if(i==3) break;
    else printf ("%d\n",i);
```

This example prints:

```
0
1
2
```

continue

Syntax

```
continue;
```

Description

The `continue` statement halts execution of its enclosing `for`, `while`, or `do/while` loop and skips to the next iteration of the loop. In the `while` and `do/while`, this means the expression is tested immediately, and in the `for` loop, the third expression (if present) is evaluated.

Example

```
/* Program name is "continue_example". This program
 * reads a file of student names and test scores. It
 * averages each student's grade. The for loop uses
 * a continue statement so that the third test score
 * is not included.
 */
#include <stdio.h>

int main(void)
{
    int test_score, tot_score, i;
    float average;
    FILE *fp;
    char fname[10], lname[15];

    fp = fopen("grades_data", "r");
    while (!feof(fp)) /* while not end of file */
    {
        tot_score = 0;
        fscanf(fp, "%s %s", fname, lname);
        printf("\nStudent's name: %s %s\nGrades: ", fname, lname);
        for (i = 0; i < 5; i++)
        {
            fscanf(fp, "%d", &test_score);
            printf("%d ", test_score);
            if (i == 2) /* leave out this test score */
                continue;
            tot_score += test_score;
        } /* end for i */
        fscanf(fp, "\n"); /* read end-of-line at end of */
                          /* each student's data */
        average = tot_score/4.0;
        printf("\nAverage test score: %4.1f\n", average);
    }
}
```

```
    } /* end while */  
    fclose(fp);  
}
```

If you execute this program, you get the following output:

```
Student's name: Barry Quigley  
Grades: 85 91 88 100 75  
Average test score: 87.8
```

```
Student's name: Pepper Rosenberg  
Grades: 91 76 88 92 88  
Average test score: 86.8
```

```
Student's name: Sue Connell  
Grades: 95 93 91 92 89  
Average test score: 92.2
```

do...while

Syntax

```
do
    statement;
while (expression);
```

Arguments

statement A null statement, simple statement, or compound statement.

exp Any expression.

Description

The `do` statement executes statements within a loop until a specified condition is satisfied. This is one of the three looping constructions in C. Unlike the `for` and `while` loops, `do...while` performs *statement* first and then tests *expression*. If *expression* evaluates to nonzero (true), *statement* executes again, but when *expression* evaluates to zero (false), execution of the loop stops. This type of loop is always executed at least once.

Two ways to jump out of a `do...while` loop prematurely (that is, before *expression* becomes false) are the following:

- Use `break` to transfer control to the first statement following the `do...while` loop.
- Use `goto` to transfer control to some labeled statement outside of the loop.

Example

```
/* Program name is "do.while_example". This program finds the
 * summation (that is,  $n*(n+1)/2$ ) of an integer that a user
 * supplies and the summation of the squares of that integer.
 * The use of the do/while means that the code inside the loop
 * is always executed at least once.
 */
#include <stdio.h>
int main(void)
{
    int num, sum, square_sum;
    char answer;

    printf("\n");
    do
    {
        printf("Enter an integer: ");
        scanf("%d", &num);
        sum = (num*(num+1))/2;
        square_sum = (num*(num+1)*(2*num+1))/6;
        printf("The summation of %d is: %d\n", num, sum);
        printf("The summation of its squares is: %d\n",
            square_sum);
        printf("\nAgain? ");
        fflush(stdin);
        scanf("%c", &answer);
    } while ((answer != 'n') && (answer != 'N'));
}
```

If you execute this program, you get the following output:

```
Enter an integer: 10
The summation of 10 is: 55
The summation of its squares is: 385

Again? y
Enter an integer: 25
The summation of 25 is: 325
The summation of its squares is: 5525

Again? n
```

for

Syntax

```
for ([expression1]; [expression2]; [expression3])  
    statement;
```

Arguments

- expression1* *Expression1* is the *initialization expression* that typically specifies the initial values of variables. It is evaluated only once before the first iteration of the loop.
- expression2* *Expression2* is the *controlling expression* that determines whether or not to terminate the loop. It is evaluated before each iteration of the loop. If *expression2* evaluates to a nonzero value, the loop body is executed. If it evaluates to 0, execution of the loop body is terminated and control passes to the first statement after the loop body. This means that if the initial value of *expression2* evaluates to zero, the loop body is never executed.
- expression3* *Expression3* is the *increment expression* that typically increments the variables initialized in *expression1*. It is evaluated after each iteration of the loop body and before the next evaluation of the controlling expression.

Description

The `for` statement executes the statement(s) within a loop as long as *expression2* is true. The `for` statement is a general-purpose looping construct that allows you to specify the initialization, termination, and increment of the loop. The `for` uses three expressions. Semicolons separate the expressions. Each expression is optional, but you must include the semicolons.

How the for Loop is Executed

The `for` statement works as follows:

1. First, *expression1* is evaluated. This is usually an assignment expression that initializes one or more variables.
2. Then *expression2* is evaluated. This is the conditional part of the statement.
3. If *expression2* is false, program control exits the `for` statement and flows to the next statement in the program. If *expression2* is true, *statement* is executed.
4. After *statement* is executed, *expression3* is evaluated. Then the statement loops back to test *expression2* again.

for Loop Processing

The `for` loop continues to execute until *expression2* evaluates to 0 (false), or until a branch statement, such as a `break` or `goto`, interrupts loop execution.

If the loop body executes a `continue` statement, control passes to *expression3*. Except for the special processing of the `continue` statement, the `for` statement is equivalent to the following:

```
expression1;  
while (expression2) {  
    statement  
    expression3;  
}
```

You may omit any of the three expressions. If *expression2* (the controlling expression) is omitted, it is taken to be a nonzero constant.

for versus while Loops

Note that `for` loops can be written as `while` loops, and vice versa. For example, the `for` loop

```
for (j = 0; j < 10; j++)  
{  
    do_something();  
}
```

is the same as the following `while` loop:

Statements

for

```
j = 0;
while (j<10)
{
    do_something();
    j++;
}
```

Example

```
/* Program name is "for_example". The following computes a
 * permutation that is,  $P(n,m) = n!/(n-m)!$  using for
 * loops to compute  $n!$  and  $(n-m)!$ 
 */
#include <stdio.h>
#define SIZE 10

int main(void)
{
    int n, m, n_total, m_total, perm, i, j, mid, count;

    printf("Enter the numbers for the permutation (n things ");
    printf("taken m at a time)\nseparated by a space: ");
    scanf("%d %d", &n, &m);
    n_total = m_total = 1;
    for (i = n; i > 0; i--) /* compute n! */
        n_total *= i;
    for (i = n - m; i > 0; i--) /* compute (n-m)! */
        m_total *= i;
    perm = n_total/m_total;
    printf("P(%d,%d) = %d\n\n", n, m, perm);

    /* This series of for loops prints a pattern of "Z's" and shows
     * how loops can be nested and how you can either increment or
     * decrement your loop variable. The loops also show the proper
     * placement of curly braces to indicate that the outer loops
     * have multiple statements.
     */
    printf("Now, print the pattern three times:\n\n");
    mid = SIZE/2;

    /* controls how many times pattern is printed */
    for (count = 0; count < 3; count++)
    {
        for (j = 0; j < mid; j++)
        {
            /* loop for printing an individual line */
            for (i = 0; i < SIZE; i++)
                if (i < mid - j || i > mid + j)
                    printf(" ");
                else
                    printf("Z");
            printf("\n");
        }
        for (j = mid; j >= 0; j--)
        {
            for (i = 0; i <= SIZE; i++)
                if (i < mid - j || i > mid + j)
```

```
        printf(" ");  
    else  
        printf("Z");  
    printf("\n");  
    }  
}
```

If you execute this program, you get the following output:

Enter the numbers for the permutation (n things taken m at a time) separated by a space: **4 3**
P(4,3) = 24

Now, print the pattern three times:

```
    Z  
    ZZZ  
    ZZZZZ  
    ZZZZZZZ  
    ZZZZZZZZZ  
    ZZZZZZZZZZZ  
    ZZZZZZZZZZZ  
    ZZZZZZZ  
    ZZZZ  
    ZZZ  
    Z  
    Z  
    ZZZ  
    ZZZZZ  
    ZZZZZZZ  
    ZZZZZZZZZ  
    ZZZZZZZZZZZ  
    ZZZZZZZZZZZ  
    ZZZZZZZ  
    ZZZZZ  
    ZZZ  
    Z  
    Z  
    ZZZ  
    ZZZZZ  
    ZZZZZZZ  
    ZZZZZZZZZ  
    ZZZZZZZZZZZ  
    ZZZZZZZZZZZ  
    ZZZZZZZ  
    ZZZZZ  
    ZZZ  
    Z
```

goto

Syntax

`goto label;`

Arguments

label This is a name or tag associated with an executable statement.

Description

The purpose of the `goto` statement is to enable program control to jump to some other statement. The destination statement is identified by a statement label, which is just a name followed by a colon. The label must be in the same function as the `goto` statement that references it.

Few programming statements have produced as much debate as the `goto` statement. The `goto` statement is necessary in more rudimentary programming languages, but its use in high-level languages is generally frowned upon. Nevertheless, most high-level languages, including C, contain a `goto` statement for those rare situations where it can't be avoided.

With deeply nested logic there are times when it is cleaner and simpler to bail out with one `goto` rather than backing out of the nested statements. The most common and accepted use for a `goto` is to handle an extraordinary error condition.

Example

The following example shows a `goto` that can easily be avoided by using the `while` loop, and also shows an illegal `goto`:

```
/* Program name is "goto_example". This program finds the
 * circumference and area of a circle when the user gives
 * the circle's radius.
 */
#include <stdio.h>
#define PI 3.14159

int main(void)
```

```

{
    float cir, radius, area;
    char answer;
    extern void something_different(void);

circles:
    printf("Enter the circle's radius: ");
    scanf("%f", &radius);
    cir = 2 * PI * radius;
    area = PI * (radius * radius);
    printf("The circle's circumference is: %6.3f\n", cir);
    printf("Its area is: %6.3f\n", area);
    printf("\nAgain? y or n: ");
    fflush(stdin);
    scanf("%c", &answer);

    if (answer == 'y' || answer == 'Y')
        goto circles;
    else {
        printf("Do you want to try something different? ");
        fflush(stdin);
        scanf("%c", &answer);
        if (answer == 'y' || answer == 'Y')
            /* goto different;          WRONG! This label is in */
            /*                          another block.          */
            something_different();
        } /* end else */
}

void something_different(void)
{
different:
    printf("Hello. This is something different.\n");
}

```

If you execute this program, you get the following output:

```

Enter the circle's radius: 3.5
The circle's circumference is: 21.991
Its area is: 38.484

Again? y or n: y
Enter the circle's radius: 6.1
The circle's circumference is: 38.327
Its area is: 116.899

Again? y or n: n
Do you want to try something different? y
Hello. This is something different.

```

if

Syntax

```
if (exp)          /* format 1 */  
    statement
```

```
if (exp)          /* format 2 */  
    statement1  
else  
    statement2
```

Arguments

exp Any expression.

statement Any null statement, simple statement, or compound statement. A statement can itself be another `if` statement. Remember, a statement ends with a semicolon.

Description

The `if` statement tests one or more conditions and executes one or more statements according to the outcome of the tests. The `if` and `switch` statements are the two conditional branching statements in C.

In the first form, if *exp* evaluates to true (any nonzero value), C executes *statement*. If *exp* is false (evaluates to 0), C falls through to the next line in the program.

In the second form, if *exp* evaluates to true, C executes *statement1*, but if *exp* is false, *statement2* is performed.

A statement can be an `if` or `if...else` statement.

Example 1

You can test multiple conditions with a command that looks like this:

```
if (exp1)          /* multiple conditions */  
    statement1  
else if (exp2)  
    statement2  
else if (exp3)  
    statement3
```

```

    .. .
    else
        statementN

```

The important thing to remember is that C executes at most only one statement in the `if...else` and `if...else/if...else` constructions. Several expressions may indeed be true, but only the statement associated with the first true expression is executed.

Example 2

Expressions subsequent to the first true expression are not evaluated. For example:

```

/* determine reason the South lost the American Civil War */
if (less_money)
    printf("It had less money than the North.\n");
else if (fewer_supplies)
    printf("It had fewer supplies than the North.\n");
else if (fewer_soldiers)
    printf("It had fewer soldiers.\n");
else
{
    printf("Its agrarian society couldn't compete with the ");
    printf("North's industrial one.\n");
}

```

All the expressions in the above code fragment could be evaluated to true, but the run-time system would only get as far as the first line and never even test the remaining expressions.

Using Braces in Compound if Statements

Use curly braces (`{ }`) in a compound statement to indicate where the statement begins and ends. For example:

```

if (x > y) {
    temp = x;
    x = y;
    y = temp;
}
else
    /* make next comparison */

```

Braces also are important when you nest `if` statements. Since the `else` portion of the statement is optional, you may not have one for an inner `if`. However, C associates an `else` with the closest previous `if` statement unless you use braces to show that isn't what you want. For example:

```

if (month == 12) {          /* month = December */
    if (day == 25)
        printf("Today is Christmas.\n");
}

```

Statements

if

```
}
else
    printf("It's not even December.\n");
```

Without the braces, the `else` would be associated with the inner `if` statement, and so the no-December message would be printed for any day in December except December 24. Nothing would be printed if `month` did not equal 12.

The Dangling else

Nested `if` statements create the problem of matching each `else` phrase to the right `if` statement. This is often called the dangling `else` problem; the general rule is:

- An `else` is always associated with the nearest previous `if`.

Each `if` statement, however, can have only one `else` clause. It is important to format nested `if` statements correctly to avoid confusion. An `else` clause should always be at the same indentation level as its associated `if`. However, don't be misled by indentations that look right even though the syntax is incorrect.

Example

```
/* Program name is "if.else_example". */
#include <stdio.h>
int main(void)
{
    int age, of_age;
    char answer;
    /* This if statement is an example of the second form (see
     * "Description" section). */
    printf("\nEnter an age: ");
    scanf("%d", &age);
    if (age > 17)
        printf("You're an adult.\n");
    else {
        of_age = 18 - age;
        printf("You have %d years before you're an adult.\n",
              of_age);
    } /* end else */
    printf("\n");
    printf("This part will help you decide whether to jog \
today.\n");
    printf("What is the weather like?\n");
    printf("    raining = r\n");
    printf("    cold = c\n");
    printf("    muggy = m\n");
    printf("    hot = h\n");
    printf("    nice = n\n");
    printf("Enter one of the choices: ");
    fflush(stdin);
```



```
scanf("\n%c", &answer);
/* This if statement is an example of the third form (see
 * "Description" section. */
if (answer == 'r')
    printf("It's too wet to jog today. Don't bother.\n");
else if (answer == 'c')
    printf("You'll freeze if you jog today. Stay indoors.\n");
else if (answer == 'm')
    printf("It's no fun to run in high humidity. Skip it.\n");
else if (answer == 'h')
    printf("You'll die of the heat if you try to jog today. \
So don't.\n");
else if (answer == 'n')
    printf("You don't have any excuses. You'd better go \
run.\n");
else
    printf("You didn't give a valid answer.\n");
}
```

If you execute this program, you get the following output:

```
Enter an age: 15
You have 3 years before you're an adult.
```

```
This part will help you decide whether to jog today.
What is the weather like?
    raining = r
    cold = c
    muggy = m
    hot = h
    nice = n
Enter one of the choices: r
It's too wet to jog today. Don't bother.
```

return

Syntax

```
return;  
return exp;          /* first form   */  
                    /* second form  */
```

Arguments

exp Any valid C expression.

Description

The `return` statement causes a C program to exit from the function containing the `return` and go back to the calling block. It may or may not have an accompanying *exp* to evaluate. If there is no *exp*, the function returns an unpredictable value.

A function may contain any number of return statements. The first one encountered in the normal flow of control is executed, and causes program control to be returned to the calling routine. If there is no `return` statement, program control returns to the calling routine when the right brace of the function is reached. In this case, the value returned is undefined.

Return Types

The return value must be assignment-compatible with the type of the function. This means that the compiler uses the same rules for allowable types on either side of an assignment operator to determine allowable return types. For example, if `f()` is declared as a function returning an `int`, it is legal to return any arithmetic type, since they can all be converted to an `int`. It would be illegal, however, to return an aggregate type or a pointer, since these are incompatible types.

The following example shows a function that returns a `float`, and some legal return values.

```
float f(void)  
{  
    float f2;  
    int a;  
    char c;
```

```

    f2 = a;      /* OK, quietly converts a to float */
    return a;   /* OK, quietly converts a to float */
    f2 = c;     /* OK, quietly converts c to float */
    return c;   /* OK, quietly converts c to float */
}

```

Pointer Return Types

The C language is stricter about matching pointers. In the following example, `f()` is declared as a function returning a pointer to a `char`. Some legal and illegal return statements are shown.

```

char *f(void)
{
    char **cpp, *cp1, *cp2, ca[10];
    int *ip1, *ip2;

    cp1 = cp2;      /* OK, types match */
    return cp2;    /* OK, types match */
    cp1 = *cpp;    /* OK, types match */
    return *cpp;   /* OK, types match */

/* An array name without a subscript is converted
 * to a pointer to the first element.
 */
    cp1 = ca;      /* OK, types match */
    return ca;    /* OK, types match */

    cp1 = *cp2;   /* Error, mismatched types          */
                /* (pointer to char vs. char)       */
    return *cp2; /* Error, mismatched types          */
                /* (pointer to char vs. char)       */
    cp1 = ip1;   /* Error, mismatched pointer types */
    return ip1; /* Error, mismatched pointer types */
    return;     /* Produces undefined behavior   */
                /* should return (char *)      */
}

```

Note in the last statement that the behavior is undefined if you return nothing. The only time you can safely use `return` without an expression is when the function type is `void`. Conversely, if you return an expression for a function that is declared as returning `void`, you will receive a compile-time error.

Functions can return only a single value directly via the `return` statement. The `return` value can be any type except an array or function. This means that it is possible to return more than a single value indirectly by passing a pointer to an aggregate type. It is also possible to return a structure or union directly. HP C implements this by passing the structure or union by reference if the structure or union is greater than eight bytes.

Statements

return

Example

```
/* Program name is "return_example".
 * This program finds the length of a word that is entered.
 */
#include <stdio.h>

int find_length( char *string )
{
    int i;
    for (i =0; string[i] != '\0'; i++);
    return i;
}

int main( void )
{
    char string[132];
    int result;
    int again = 1;
    char answer;

    printf( "This program finds the length of any word you ");
    printf( "enter.\n" );
    do
    {
        printf( "Enter the word: ");
        fflush(stdin);
        gets( string );
        result = find_length( string );
        printf( "This word contains %d characters. \n", result);
        printf("Again? ");
        scanf("%c", &answer);
    } while (answer == 'Y' || answer == 'y');
}
```

If you execute this program, you get the following output:

This program finds the length of any string you enter.

```
Enter the string: Copenhagen
The string is 10 characters.
Again? y
```

```
Enter the string: galaxy
The string is 6 characters.
Again? n
```

switch

Syntax

```
switch ( exp )
{
  case const_exp : [statement]...
  [case const_exp : [statement]...]
  [default : [statement]...]
}
```

Arguments

<i>exp</i>	The integer expression that the <code>switch</code> statement evaluates and then compares to the values in all the cases.
<i>const_exp</i>	An integer expression to which <i>exp</i> is compared. If <i>const_exp</i> matches <i>exp</i> , the accompanying statement is executed.
<i>statement</i>	Zero or more simple statements. (If there is more than one simple statement, you do not need to enclose the statements in braces.)

Description

The `switch` statement is a conditional branching statement that selects among several statements based on constant values.

The expression immediately after the `switch` keyword must be enclosed in parentheses and must be an integral expression.

The expressions following the `case` keywords must be integral constant expressions; that is, they may not contain variables.

An important feature of the `switch` statement is that program flow continues from the selected `case` label until another control-flow statement is encountered or the end of the `switch` statement is reached. That is, the compiler executes any statements following the selected case label until a `break`, `goto`, or `return` statement appears. The `break` statement explicitly exits the `switch` construct, passing control

Statements

switch

to the statement following the `switch` statement. Since this is usually what you want, you should almost always include a `break` statement at the end of the statement list following each case label.

The following `print_error()` function, for example, prints an error message based on an error code passed to it.

```
/* Prints error message based on error_code.
 * Function is declared with void because it doesn't
 * return anything.
 */
#include <stdio.h>
#define ERR_INPUT_VAL 1
#define ERR_OPERAND 2
#define ERR_OPERATOR 3
#define ERR_TYPE 4
void print_error(int error_code)
{
    switch (error_code) {
        case ERR_INPUT_VAL:
            printf("Error: Illegal input value.\n");
            break;
        case ERR_OPERAND:
            printf("Error: Illegal operand.\n");
            break;
        case ERR_OPERATOR:
            printf("Error: Unknown operator.\n");
            break;
        case ERR_TYPE:
            printf("Error: Incompatible data.\n");
            break;
        default: printf("Error: Unknown error code %d\n",
                       error_code);
            break;
    }
}
```

The `break` statements are necessary to prevent the function from printing more than one error message. The last `break` after the default case is not really necessary, but it is a good idea to include it anyway for the sake of consistency.

Evaluation of switch Statement

The `switch` expression is evaluated; if it matches one of the `case` labels, program flow continues with the statement that follows the matching `case` label. If none of the `case` labels match the `switch` expression, program flow continues at the default label, if it exists. (The default label need not be the last label, though it is good style to put it last.) No two `case` labels may have the same value.

Associating Statements with Multiple case Values

Sometimes you want to associate a group of statements with more than one case value. To obtain this behavior, you can enter consecutive case labels. The following function, for instance, returns 1 if the argument is a punctuation character, or 0 if it is anything else.

```
/* This function returns 1 if the argument is a
 * punctuation character. Otherwise, it returns 0.
 */
is_punc(char arg)
{
    switch (arg) {
        case '.':
        case ',':
        case ':':
        case ';':
        case '?':
        case '-':
        case '(':
        case ')':
        case '!': return 1;
        default : return 0;
    }
}
```

Example

```
/* Use the switch statement to decide which comment should be
printed */
#include <stdio.h>
int main(void)
{
    char answer, grade;
    answer = 'y';
    printf("\n\n");
    while (answer == 'y' || answer == 'Y') {
        printf("Enter student's grade: ");
        fflush(stdin);
        scanf("%c", &grade);
        printf("\nComments: ");
        switch (grade) {
            case 'A':
            case 'a':
                printf("Excellent\n");
                break;
            case 'B':
            case 'b':
                printf("Good\n");
                break;
            case 'C':
            case 'c':
                printf("Average\n");
                break;
            case 'D':
            case 'd':
```

Statements

switch

```
        printf("Poor\n");
        break;
    case 'E':
    case 'e':
    case 'F':
    case 'f':
        printf("Failure\n");
        break;
    default:
        printf("Invalid grade\n");
        break;
} /* end switch */
printf("\nAgain? ");
fflush(stdin);
scanf("%s", &answer);
}
}
```

If you execute this program, you get the following output:

Enter student's grade: **B**

Comments: Good

Again? **y**

Enter student's grade: **C**

Comments: Average

Again? **n**

while

Syntax

```
while ( exp )  
    statement
```

Arguments

exp Any expression.

statement This statement is executed when the `while` (*exp*) is true.

Description

The `while` statement executes the statements within a loop as long as the specified condition, *exp*, is true. This is one of the three looping constructions available in C. Like the `for` loop, the `while` statement tests *exp* and if it is true (nonzero), *statement* is executed. Once *exp* becomes false (0), execution of the loop stops. Since *exp* could be false the first time it is tested, *statement* may not be performed even once.

The following describes two ways to jump out of a `while` loop prematurely (that is, before *exp* becomes false):

- Use `break` to transfer control to the first statement following the `while` loop.
- Use `goto` to transfer control to some labeled statement outside the loop.

Statements

while

Example

```
/* Program name is "while_example" */
#include <stdio.h>

int main(void)
{
    int count = 0, count2 = 0;
    char a_string[80], *ptr_to_a_string = a_string;

    printf("Enter a string -- ");
    gets(a_string);

    while (*ptr_to_a_string++)
        count++; /* A simple statement loop */
    printf("The string contains %d characters.\n", count);
    printf("The first word of the string is ");

    while (a_string[count2] != ' ' && a_string[count2] != '\0')
    {
        /* A compound statement loop */
        printf ("%c", a_string[count2]);
        count2++;
    }
    printf("\n");
}
```

If you execute this program, you get the following output:

```
Enter a string Four score and seven years ago
The string contains 30 characters.
The first word of the string is Four
```

7

Preprocessing Directives

Preprocessing directives function as compiler control lines. They allow you to direct the compiler to perform certain actions on the source file.

You can select from any of the following topics:

- Source File Inclusion (`#include`)
- Macro Replacement (`#define`, `#undef`)
- Predefined Macros
- Conditional Compilation (`#if`, `#ifdef`, `..#endif`)
- Line Control (`#line`)
- Pragma Directive (`#pragma`)
- Error Directive (`#error`)
- Trigraph Sequences

Overview of the Preprocessor

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 the text. It also expands preprocessor macros and conditionally strips out comments.

Syntax

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

Description

The preprocessing directives control the following general functions:

1. Source File Inclusion

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

2. Macro Replacement

You can direct the compiler to replace token sequences with other token sequences. This is frequently used to define names for constants rather than hard coding them into the source files.

3. Conditional Inclusion

You can direct the compiler to check values and flags, and 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 computers.

4. Line Control

You can direct the compiler to increment subsequent lines from a number specified in a control line.

5. Pragma Directive

Pragmas are implementation-dependent instructions that are directed to the compiler. Because they are very system dependent, they are not portable.

All preprocessing directives begin with a pound sign (#) as the first character in a line of a source file. White space may precede the # character in preprocessing directives. The # character is followed by any number of spaces and horizontal tab characters and the preprocessing directive. The directive is terminated by a new-line character. You can continue directives, as well as normal source lines, over several lines by ending lines that are to be continued with a backslash (\).

Comments in the source file that are not passed through the preprocessor are replaced with a single white-space character.

Examples

```
include-directive:    #include <stdio.h>
macro-directive:    #define MAC x+y
conditional-directive: #ifdef MAC
line-directive:     #line 5 "myfile"
pragma-directive:  #pragma INTRINSIC func
```

Source File Inclusion (#include)

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

Syntax

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

Description

In the third form above, *identifier* must be in the form of one of the first two choices after macro replacement.

The `#include` preprocessing directive causes the compiler to switch its input file so that source is taken from the file named in the include directive. Historically, include files are named:

```
filename.h
```

If the file name is enclosed in double quotation marks, the compiler searches your current directory for the specified file. If the file name is enclosed in angle brackets, the “system” directory is searched to find the named file. Refer to Chapter 10 for a detailed description of how the directory is searched.

Files that are included may contain `#include` directives themselves. The HP C compiler 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. Error messages produced by the HP C compiler usually supply the file name the error occurred in as well as the file relative line number of the error.

Examples

```
#include <stdio.h>
#include "myheader"

#ifdef MINE
#   define filename "file1"
#else
#   define filename "file2"
#endif

#include filename
```

Macro Replacement (#define, #undef)

You can define text substitutions in your source file with C macro definitions.

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 or 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 translation unit.

Macros can be redefined without an intervening #undef directive. Any parameters used must agree in number and spelling, and the replacement lists must be identical. All white space is treated equally.

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

If the define takes the form

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

a macro with formal parameters is defined. The macro name is the *identifier* and the formal parameters are provided by the *identifier-list* which is enclosed in parentheses. The first parenthesis must immediately follow the identifier with no intervening white space. If

there is a space between the identifier and the (, the macro is defined as if it were the first form and that 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 parentheses-enclosed list following the macro name. Comma tokens 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.

If a formal parameter in the macro definition directive's token string follows 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 may be used to turn macro arguments into strings. This feature is often used with the fact that the compiler concatenates adjacent strings.

After all replacements have taken place during macro invocation, each instance of the special ## token is deleted and the tokens preceding and following the ## are concatenated into a single token. This is useful in forming unique variable names within macros.

The following example illustrates the use of the # operator for creating string literals out of arguments and concatenating tokens:

```
#define debug(s, t) printf("x" # s "= %d, x" # t " %s", x ## s, x ## t)
```

Invoked as: `debug(1, 2);`

Results in:

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
```

which, after concatenation, results in:

```
printf("x1= %d, x2= %s", x1, x2);
```

Spaces around the # and ## are optional.

NOTE

The # and ## operators are only supported in ANSI mode.

Preprocessing Directives

Macro Replacement (#define, #undef)

The most common use of the macro replacement is in defining a constant. Rather than hard coding constants in a program, you can name the constants using macros then use the names in place of actual 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 will change and so will all the expressions that make use of the dimension.

Some other common macros used by C programmers include:

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

The following macro is more complex. It has two parameters and will produce an in-line expression which is equal to the maximum of its two parameters:

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

Parentheses surrounding each argument and the resulting expression insure that the precedences of the arguments and the result will not improperly interact with any other operators that might be used with the `MAX` macro.

Using a macro definition for `MAX` has some advantages over a function definition. First, it executes faster because the macro generates in-line code, avoiding the overhead of a function call. Second, the `MAX` macro accepts any argument types. A functional implementation of `MAX` would be restricted to the types defined for the function. Note further that because each argument to the `MAX` macro appears in the token string more than once, check to be sure that the actual arguments to the `MAX` macro do not have any "side effects." The following example

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

might not work as expected because the argument `a` is incremented two times when `a` is the maximum.

The following statement

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

is expanded to:

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

Examples

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

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

Predefined Macros

In addition to `__LINE__` and `__FILE__` (see “Line Control (#line)” on page 190), ANSI C provides the `__DATE__`, `__TIME__` and `__STDC__` predefined macros. Table 7-1 describes the complete set of macros that are predefined to produce special information. They may not be undefined.

Table 7-1 **Predefined Macros**

Macro Name	Description
<code>__DATE__</code>	Produces the date of compilation in the form Mmm dd yyyy.
<code>__FILE__</code>	Produces the name of the file being compiled.
<code>__LINE__</code>	Produces the current source line number.
<code>__STDC__</code>	Produces the decimal constant 1, indicating that the implementation is standard-conforming.
<code>__TIME__</code>	Produces the time of compilation in the form hh:mm:ss.

NOTE `__DATE__`, `__TIME__`, and `__STDC__` are only defined in ANSI mode.

Conditional Compilation (#if, #ifdef, ..#endif)

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

Syntax

```
conditional-directive ::=  
    #if      constant-expression newline [group]  
    #ifdef   identifier newline [group]  
    #ifndef  identifier newline [group]  
    #else    newline [group]  
    #elif    constant-expression newline [group]  
    #endif
```

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 structure of the conditional compilation directives can be shown using the `#if` directive:

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

Preprocessing Directives

Conditional Compilation (`#if`, `#ifdef`, `..#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, or an enumeration constant.

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. For 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
#ifdef min
```

The `#if` preprocessing directive has the form:

```
#if constant-expression
```

Use `#if` to test an expression. The compiler evaluates the expression in the directive. If it is true (a nonzero value), the code following the directive is included. If the expression evaluates to false (a zero value), the compiler 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.

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

The following preprocessing directives are used to 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.

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

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 compiled if the code following any of the if directives does not compile.

The #elif *constant-expression* directive tests whether a condition of the previous #if, #ifdef, or #ifndef was false. #elif is syntactically the same as the #if directive and can be used in place of an #else directive.

Examples

Valid combinations of these conditional compilation directives follow:

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

#ifdef THING
/* compiled if THING is defined */
#endif
/* end of if */

#if A>47
/* compiled if A evaluates > 47 */
#else
#if A < 20
/* compiled if A evaluates < 20 */
#else
/* compiled if A >= 20 and <= 47 */
#endif
/* end of if, A < 20 */
#endif
/* end of if, A > 47 */
```

Examples

```
#ifdef (HP9000_S800) /* If HP9000_S800 is defined, INT_SIZE */
#define INT_SIZE 32 /* is defined to be 32 (bits). */
#elif defined (HPVECTRA) && defined (SMALL_MODEL)
#define INT_SIZE 16 /* Otherwise, if HPVECTRA and */
#endif /* SMALL_MODEL are defined,INT_SIZE is */

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

Line Control (#line)

You can cause the compiler to increment 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 the compiler to treat lines following it in the program as if the name of the source file were *filename* and the current line number is *digit-sequence*. This is to control the file name and line number that is given in diagnostic messages, for example. This feature is used primarily for preprocessor programs that generate C code. It enables them to force the HP C compiler 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 and subsequently input to the compiler.

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. Note that you can change

`__FILE__` and `__LINE__` using #include or #line directives.

Example

```
#line digit-sequence [filename]: #line 5 "myfile"
```


Pragma Directive (#pragma)

You can provide instructions to the compiler through inclusion of pragmas.

Syntax

```
pragma-directive ::=  
#pragma replacement-list
```

Description

The #pragma preprocessing directive provides implementation-dependent information to the compiler. See Chapter 9, “Compiling and Running HP C Programs,” on page 197 for descriptions of pragmas recognized by HP C/HP-UX. Any pragma that is not recognized by the compiler is ignored.

Example

```
#pragma replacement-list: #pragma intrinsic func
```

Error Directive (#error)

Syntax

```
#error [pp-tokens]
```

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

Examples

```
#ifndef (HP_C)
#error "HP_C not defined!"      /* This directive will produce
                              the diagnostic message "HP_C
                              not defined!" */
#endif

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

NOTE

The `#error` directive is only supported in ANSI mode.

Trigraph Sequences

The C source code character set is a superset of the ISO 646-1983 Invariant Code Set. To enable programs to be represented in the reduced set, *trigraph sequences* are defined to represent those characters not in the reduced set. A *trigraph* is a three character sequence that is replaced by a corresponding single character. Table 7-2 gives the complete list of trigraph sequences and their replacement characters.

Table 7-2 **Trigraph Sequences and Replacement Characters**

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

Any ? that does not begin one of the trigraphs listed above is not changed.

Preprocessing Directives
Trigraph Sequences

8

C Library Functions

The C library (`/usr/lib/libc.a` or `/lib/libc.sl`) is divided into different subsections. Each subsection has a header file that defines the objects found in that section of the library.

The standard headers are:

```
<assert.h>      <locale.h>      <stddef.h>
<ctype.h>       <math.h>         <stdio.h>
<errno.h>       <setjmp.h>       <stdlib.h>
<float.h>       <signal.h>       <string.h>
<limits.h>      <stdarg.h>       <time.h>
```

The order of inclusion of these header files using the `#include` directive makes no difference. Also, if you include the same header file more than once, an error does not occur.

Function names beginning with an underscore (`_`) are reserved for library use; you should not specify identifiers that begin with an underscore.

To use some facilities, the C source code must include the preprocessor directive:

```
#include <libraryname.h>
```

The preprocessor looks for the particular header file defined in *libraryname* in a standard location on the system.

The standard location is `/usr/include`.

The *libraryname* must be enclosed in angle brackets. For example, if you want to use the `fprintf` function, which is in the standard I/O library, your program must specify

```
#include <stdio.h>
```

because the definition of `fprintf`, as well as various types and variables used by the I/O function, are found in the `stdio.h` header file.

C Library Functions

The C library contains both functions and macros. The use of macros improves the execution speed of certain frequently used operations. One drawback to using macros is that they do not have an address. For example, if a function expects the address of (a pointer to) another function as an argument, you cannot use a macro name in that argument. The following example illustrates the drawback:

```
#define add1(x) ((x)+=1)

extern f();

main()
{
    .
    .
    f(add1); <This construct is illegal.
    .
    .
}
```

Using `add1` as an argument causes an error.

The `#undef` directive may be used to reference a true function instead of a macro.

There are three ways in which a function can be used:

- In a header file (which might generate a macro)

```
#include <string.h>
i = strlen(x);
```

- By explicit declaration

```
extern int strlen();
i=strlen(x);
```

- By implicit declaration

```
i = strlen(x);
```

For more information on C library functions, see the *HP-UX Reference* and *HP-UX Linker and Libraries Online User Guide*.

9

Compiling and Running HP C Programs

This chapter describes how to compile and run HP C programs on HP 9000 systems. The compiler command and its options are presented. You can compile HP C programs to assembly, object, or executable files. You can also optionally optimize code to improve application run-time speed.

Compiling HP C Programs

When you compile a program, it passes through one or more of the following steps depending upon which command line options you use:

- **Preprocessor:** This phase examines all lines beginning with a # and performs the corresponding actions and macro replacements.
- **Compilation Process:** This phase takes the output of the preprocessor and generates object code.
- **Optimization:** This optional phase optimizes the generated object code.
- **Linking:** In this phase, the linker is invoked to produce an executable program. External references in shared and archived libraries are resolved as required. The startup routines in `/opt/langtools/lib/crt0.o` are copied in, and the C library in `/lib/libc.sl` or `/lib/libc.a` is referenced. (By default, shared libraries take precedence over archived libraries if both versions are available. However, if you use the `LPTH` environment variable, you should make sure that all shared libraries come before any archive library directories. See the *HP-UX Linker and Libraries Online User Guide* for information on `LPTH` or on creating and linking with shared libraries.) Object modules are combined into an executable program file.

If you are referencing 64-bit binaries on the compile-line you must specify the `+DD64` option, or the compiler will pass the 32-bit `crt0.o` to the linker, producing a Mismatched ABI error message from the linker.

Compatibility Mode vs. ANSI C Mode

The HP C compiler provides two modes of operation: *compatibility mode* and *ANSI C mode*. The compatibility mode option causes the compiler to compile code in a manner similar to version 3.1. ANSI C mode is a strict implementation of the standard. See “HP C Compiler Options” on page 200 for more information.

The cc(1) Command

Use the `cc(1)` command to compile HP C programs. It has the following format:

```
cc [options] files
```

where:

options is one or more compiler options and their arguments, if any. Options can be grouped together under one minus sign.

files is one or more file names, separated by blanks. Each file is either a source or an object file.

Specifying Files to the cc Command

Files with names ending in `.c` are assumed to be HP C source files. Each HP C source file is compiled, producing an object file with the same name as the source file except that the `.c` extension 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.

Files with names ending in `.i` are assumed to be preprocessor output files (see the `-P` compiler option in Table 9-2 on page 207 in this chapter). Files ending in `.i` are processed the same as `.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 included in the linking. All other files are passed directly to the linker by the compiler.

Specifying Options to the cc Command

Each compiler option has the following format:

```
-optionname [optionarg]
```

where:

optionname is the name of a standard compiler option.

optionarg is the argument to *optionname*.

The optional argument `--` delimits the end of options. Any following arguments are treated as operands (typically input filenames) even if they begin with the minus (`-`) character.

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`.

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 name `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 `-v`, `-g`, and `-DPROG=sub` compiler options. There are several ways to do this:

```
cc my_file.c -v -g -DPROG=sub
cc my_file.c -vg -D PROG=sub

cc my_file.c -vgDPROG=sub
cc -vgDPROG=sub my_file.c
```

HP C Compiler Options

Table 9-1 summarizes the command line options supported by HP 9000 workstations and servers. See Table 9-2 for detailed information about each option.

Table 9-1 HP C Compiler Options at a Glance

Option	Description
-Aa	Enables strict ANSI C compliance.
-Ac	Disables ANSI C compliance (HP C version 3.1 compatibility).
-Ae	Enables ANSI C compliance, HP value-added features (as described for +e option), and <code>_HPUX_SOURCE</code> name space macro. It is equivalent to <code>-Aa +e -D_HPUX_SOURCE</code> .
-Bextern	Performs the same operation as <code>+Oextern=sym1,sym2,sym3...</code> except that symbols are loaded from an existing file, instead of specified on the command line.
-c	Compiles only, does not link.
-C	Prevents the preprocessor from stripping comments.
-Dname	Defines the preprocessor variable <i>name</i> with a value of "1".
-Dname=def	Defines the preprocessor variable <i>name</i> with a value of <i>def</i> .
-dynamic	Enables linking of PIC objects.
-E	Performs preprocessing only with output to <code>stdout</code> .
-fast	Expands into a set of compiler options which results in improved application run-time. Options included are <code>+O3</code> , <code>+Onooptransform</code> , <code>+Olibcalls</code> , <code>+FPD</code> , <code>+Oentryschedule</code> , and <code>+Ofastaccess</code> . Any of these options can be overridden by placing a subsequent option after <code>-fast</code> on the command line.
-g	Inserts information for the symbolic debugger in the object file.
-G	Inserts information required by the <code>gprof</code> profiler in the object file.

Compiling and Running HP C Programs
 Compiling HP C Programs

Option	Description
-I <i>dir</i>	Inserts <i>dir</i> in the include file search path.
-l <i>x</i>	Links with the <code>/lib/libx.a</code> and <code>/usr/lib/libx.a</code> libraries.
-L <i>dir</i>	Links the libraries in <i>dir</i> before the libraries in the default search path.
-n	Generates shareable code.
-N	Generates unshareable code.
-noshared	Creates statically-bound executables.
-o <i>outfile</i>	Places object modules in <i>outfile</i> file.
-O	Optimizes at level 2.
-p	Inserts information required by the <code>prof</code> profiler in the object file.
-P	Performs preprocessing only with output to the corresponding <i>.i</i> file.
-q	Marks the executable as demand loadable.
-Q	Marks the executable as not being demand loadable.
-s	Strips the symbol table from the executable file.
-S	Generates an assembly language source file.
-t <i>x,name</i>	Substitutes or inserts subprocess <i>x</i> with <i>name</i> .
-U <i>name</i>	Undefines <i>name</i> in the preprocessor.
-v	Enables verbose mode.
-V	Causes subprocesses to print version information to <code>stderr</code> .
-w	Suppresses warning messages.
-Wd, -a or +a	Omits HP provided prefix files required by the linker.

Option	Description
-W <i>x</i> , <i>arg1</i> [<i>arg2</i> ,..., <i>argn</i>]	Passes the arguments <i>arg1</i> through <i>argn</i> to the subprocess <i>x</i> .
-y	Generates information used by the <i>HP SoftBench</i> static analysis tool.
-Y	Enables Native Language Support (NLS).
-z	Disallows run-time dereferencing of null pointers.
-Z	Allows dereferencing of null pointers at run-time.
+D <i>Amodel</i>	Generates object code for a specific version of the PA-RISC architecture.
+DC <i>apptype</i>	Generate code for portable or embedded applications.
+DD32	Generates 32-bit code for PA1.1 architecture.
+DD64	Generates 64-bit code for PA2.0 architecture.
+df <i>name</i>	Specifies the profile database to use with profile-based optimization.
+DS <i>model</i>	Performs instruction scheduling for a specific implementation of PA-RISC.
+DO <i>osname</i>	Sets the target operating system for the compiler.
+e	Enables the following HP value added features while compiling in ANSI C mode: <code>sized enum</code> , <code>long long</code> , <code>long pointers</code> , <code>compiler supplied defaults for missing arguments to intrinsic calls</code> , and <code>\$</code> in identifier HP C extensions.
+ES <i>fic</i>	Compiles with inline fast indirect calls.
+ES <i>constlit</i>	Introduces new default behavior. HP C now stores constant-qualified (<code>const</code>) objects and literals in read-only memory.
+ES <i>lit</i>	Places string literals and constants into read-only data storage.

Compiling and Running HP C Programs
Compiling HP C Programs

Option	Description
+ESnolit	Disables +ESconstlit, causing HP C to no longer store literals in read-only memory.
+ESnoparmreloc	Disables parameter relocation for function calls.
+ESsfc	Replaces function pointer comparison millicode calls with inline code.
+f	Inhibits the promotion of float to double, except for function calls and returns.
+FP <i>flags</i>	Controls floating-point traps.
+help	Launches a web browser displaying an html version of the HP C/HP-UX Online Help.
+hugesize	Lowers the threshold for huge data.
+I	Prepares the object code for profile-based optimization data collection.
+k	Generates long-displacement code sequences so a program can reference large amounts of global data physically located in shared libraries.
+L	Enables any #pragma listing directives and the listing facility.
+m	Prints identifier maps in the source code listing.
+M	Provides ANSI migration warnings that explain the differences between code compiled with -Ac and -Aa.
+M1	Provides platform migration warnings for features that may not be supported in future releases.
+M2	Provides migration warnings for transitioning code from the ILP32 to the LP64 data model.
+o	Prints hexadecimal code offsets in the source code listing.

Option	Description
+[no]objdebug	When used with -g, +objdebug leaves debug information in the object files instead of copying it to the executable file. The object files must be accessible to the HP WDB debugger when debugging. This option is not supported by the HP DDE debugger.
+O <i>opt</i>	Invokes optimization level <i>opt</i> where <i>opt</i> is 0 to 4. See the <i>HP C/HP-UX Programmer's Guide</i> for additional optimization options.
+P	Performs profile-based optimization.
+pa	Requests routine-level profiling with CXperf.
+pal	Requests routine-level and loop-level profiling with CXperf.
+pgm <i>name</i>	Specifies the execution profile data set to be used by the optimizer.
+r	Inhibits the automatic promotion of float to double.
+sb	Make bit-fields signed by default in both 32- and 64-bit modes.
+u <i>bytes</i>	Controls pointer alignment where <i>bytes</i> is 1, 2, or 4.
+uc	Makes unqualified char data types unsigned.
+wn	Specifies the level of the warning messages where <i>n</i> is 1 - 3.
+W <i>n1</i> [, <i>n2</i> ,... <i>nN</i>]	Suppresses the specified warnings.
+We <i>n1</i> [, <i>n2</i> ,... <i>nN</i>]	Changes the specified warnings to errors.

Compiling and Running HP C Programs
Compiling HP C Programs

Option	Description
+Ww n1[,n2,...nN]	Enables the specified warnings, assuming all other warnings are suppressed with -w or +w3.
+z	Generates shared library object code (same as +Z in 64-bit mode).
+Z	Generates shared library object code with a large data linkage table (long-form PIC).

Table 9-2 describes in detail the options that HP C workstations and servers support.

Table 9-2 HP C Compiler Option Details

Option	What It Does
-Aa	<p>Requests a compilation in ANSI C mode. The <code>-Aa</code> option requests a strict implementation of ANSI C. ANSI C specifies which names are available in the standard libraries and headers, which are reserved for the implementation, and which must be left available for you. HP C in ANSI mode conforms to these restrictions and only names permitted by ANSI C are defined or declared in the standard libraries and headers. Macro definitions can be used to access names that are normally defined in other standards (POSIX and XOPEN, for example), and in HP C Version 3.1 compatibility mode. To gain access to all variables and functions defined by POSIX, include the following line at the start of your source file before including any system headers:</p> <pre>#define _POSIX_SOURCE</pre> <p>or define this macro by using the <code>-D name</code> option:</p> <pre>cc -D _POSIX_SOURCE myfile.c</pre> <p>To gain access to all the names defined by XOPEN, include the following line at the start of your source file before including any system headers:</p> <pre>#define _XOPEN_SOURCE</pre> <p>or define the macro on the command line:</p> <pre>cc -D _XOPEN_SOURCE myfile.c</pre> <p>To gain access to all the names normally available in compatibility mode, include the following line at the start of your source file before including any system headers:</p> <pre>#define _HPUX_SOURCE</pre> <p>or define the macro on the command line:</p> <pre>cc -D _HPUX_SOURCE myfile.c</pre> <p><code>_HPUX_SOURCE</code> defines a superset of names defined by <code>_XOPEN_SOURCE</code>. <code>_XOPEN_SOURCE</code> defines a superset of names defined by <code>_POSIX_SOURCE</code>.</p>

Compiling and Running HP C Programs
 Compiling HP C Programs

Option	What It Does
-Ac	Requests compatibility with version 3.1. This was the default at the HP-UX 10.20 operating system release. It is a non-ANSI implementation.
-Ae	Requests a compilation in ANSI C mode with HP C extensions. This option is the same as specifying -Aa, -D_HPUX_SOURCE, and +e. This is the default for the HP-UX 10.30 and later operating system releases.
-Bextern	Performs the same operation as +Oextern=sym1,sym2,sym3... except that symbols are loaded from an existing file, instead of specified on the command line. -Bextern does not require that you manually specify symbols locally, on the command line. In fact, if you specify a locally-defined symbol with -Bextern, you may experience problems. When used, -Bextern tells the compiler that the symbols reside in a separate load module. -Bextern:filename fetches the list of symbols from a text file instead of locally. Symbols referenced in the file must be separated by white space within the file.
-c	Compiles one or more source files but does not enter the linking phase. The compiler produces the object file (a file ending with .o by default) for each source file (a file ending with .c, .s, or .i). Object files must be linked before they can be executed.
-C	Prevents the preprocessor from stripping comments. See the description of cpp(1) in the <i>HP-UX Reference Manual</i> for details.
-Dname -Dname=def	Defines <i>name</i> to the preprocessor <code>cpp</code> as if defined by the preprocessing directive <code>#define</code> . If <i>def</i> is not given, <i>name</i> is "1". Macros should be assigned integer values when they will be used in constant expressions that are to be evaluated by the conditional compilation directives <code>#if</code> and <code>#else</code> . See "Conditional Compilation (<code>#if</code> , <code>#ifdef</code> , <code>..#endif</code>)" on page 187 for more information.

Option	What It Does
-dynamic	<p>This option produces dynamically bound executables. These support shared libraries and use the features of the dynamic loader. The <i>shl_load</i> and <i>dlopen</i> APIs are fully supported. The linker links in shared libraries first and then archive libraries. Only PIC is supported.</p>
-E	<p>Runs the preprocessor only on the named HP C or assembly programs and sends the result to standard output (<i>stdout</i>).</p>
-fast	<p>The <code>-fast</code> compiler option expands into an existing set of compiler options which, used together, result in optimum speed and application run-times. This option is also provided for compatibility in porting other vendors' applications to HP-UX.</p> <p>NOTE: Do not use this option for programs that depend on IEEE standard floating-point denormalized numbers. Otherwise, different numerical results may occur.</p> <p><code>-fast</code> expands to the following HP C/ANSI C options:</p> <pre>+O3 +nolooptransform +Olibcalls +FPD +Oentrysched +Ofastaccess</pre> <ul style="list-style-type: none"> • <code>+O3</code> optimizations include all <code>+O2</code> optimizations as well as all subprograms within a single file. It also inlines certain subprograms within the input file. • <code>+Onolooptransform</code> disables transformation of eligible loops for improved cache performance. • <code>+Olibcalls</code> increases the runtime performance of code which calls the standard library routines. • <code>+FPD</code> enables a fast underflow mode on PA-7100 architectures or later where denormalized floating-point operands are flushed to zero. • <code>+Oentrysched</code> optimizes instruction scheduling on a procedure's entry and exit sequences. • <code>+Ofastaccess</code> optimizes code for fast access to global data items. <p>You can override any of the options in <code>-fast</code> by specifying a subsequent option after it.</p>

Option	What It Does
-g	<p>Inserts information for the symbolic debugger in the object file. In conjunction with the HP Distributed Debugging Environment (DDE), the C compiler now provides support for debugging optimized code. This support includes:</p> <ul style="list-style-type: none"> • Tracebacks with line-number annotation • Setting breakpoints and single-stepping at the source statement level • Mapping between source statements and machine instructions • Viewing and modifying global variables at procedure call boundaries • Viewing and modifying parameters on procedure entry <p>To enable debugging of optimized code, specify the <code>-g</code> compile-line option together with the <code>-0</code>, <code>+01</code>, or <code>+02</code> option. Currently, debugging is supported at optimization levels 2 and below. If you try to use <code>-g</code> with the <code>+03</code> or <code>+04</code> option, the compiler will issue a warning stating that the options are incompatible, and will ignore the <code>-g</code> option.</p>
-G	<p>Prepares the object file for profiling with <code>gprof</code>. See the <i>gprof(1)</i> description in the <i>HP-UX Reference</i> manual for details.</p>
-I <i>dir</i>	<p>Adds <i>dir</i> to the list of directories that are searched for include files by the preprocessor. For include files that are enclosed in double quotes and do not begin with a <code>/</code>, the preprocessor first searches the current directory, then the directory named in the <code>-I</code> option, and finally, the standard include directory <code>/usr/include</code>. For include files that are enclosed in <code><</code> and <code>></code> symbols, the search path begins with the directory named in the <code>-I</code> option and is completed in the standard include directory, <code>/usr/include</code>. The current directory is not searched.</p>

Option	What It Does
-lx	<p>Causes the linker to search the libraries <code>/usr/lib/libx.a</code> or <code>/usr/lib/libx.sl</code> (searched first) and <code>/opt/langtools/lib/libx.a</code> or <code>/opt/langtools/lib/libx.sl</code> (searched second). The <code>-a</code> linker option determines whether the archive (<code>.a</code>) or shared (<code>.sl</code>) version of a library is searched. The linker searches the shared version of a library by default.</p> <p>Libraries are searched when their names are encountered. Therefore placement of a <code>-l</code> is significant. If a file contains an unresolved external reference, the library containing the definition must be placed after the file on the command line. See the description of <code>ld(1)</code> in the <i>HP-UX Reference Manual</i> for details.</p>
-Ldir	<p>Causes the linker to search for libraries in the directory <code>dir</code> before using the default search path. The default search path is the directory <code>/usr/lib</code> followed by <code>/opt/langtools/lib</code>. <code>-Ldir</code> must precede <code>-lx</code> on the command line; otherwise <code>-Ldir</code> is ignored. This option is passed directly to the linker.</p> <p>For example:</p> <pre>cc -L/project/libs prog.c -lfoo -lbar</pre> <p>Compiles and links <code>prog.c</code> and directs the linker to search the directories <code>/project/libs</code>, <code>/usr/lib</code>, then <code>/opt/langtools/lib</code> for <code>libfoo.a</code>, <code>libfoo.sl</code>, <code>libbar.a</code>, and <code>libbar.sl</code> libraries.</p>
-n	<p>Causes the program file produced by the linker to be marked as shareable. For details and system defaults see the description of <code>ld(1)</code> in the <i>HP-UX Reference manual</i>.</p>
-N	<p>Causes the program file produced by the linker to be marked as unshareable. For details and system defaults see the <code>ld(1)</code> description in the <i>HP-UX Reference Manual</i>.</p>
-noshared	<p>Creates statically bound executables. This option is available with option <code>+DA2.0W</code> only. See <code>-dynamic</code> for dynamically bound executables. The default behavior is dynamic.</p>

Compiling and Running HP C Programs
 Compiling HP C Programs

Option	What It Does
-o <i>outfile</i>	Causes the output of the compilation sequence to be placed in <i>outfile</i> . The default name is <i>a.out</i> . When compiling a single source file with the -c option, you can also use the -o option to specify the name and location of the object file.
-O	Invokes the optimizer to perform level 2 optimization. Other optimization levels can be set. Refer to the <i>HP C/HP-UX Programmer's Guide</i> for details on optimization.
-p	Causes the compiler to produce extra profiling code that counts the number of times each routine is called. Also, if link editing takes place, this option replaces the standard startup routine with a routine that calls <code>monitor(3C)</code> at the start and writes out a <i>mon.out</i> file upon normal termination of the object program's execution. An execution profile can then be generated using <code>prof(1)</code> .
-P	Preprocesses only. Runs the preprocessor with the -P option only on the named HP C source files and leaves the result in the corresponding files with <i>.i</i> as the suffix.
-q	Causes the output file from the linker to be marked as demand loadable. For details and system defaults, see the <code>ld(1)</code> description in the <i>HP-UX Reference</i> manual.
-Q	Causes the program file created by the linker to be marked as <i>not</i> demand loadable. For details and system defaults, see the <code>ld(1)</code> description in the <i>HP-UX Reference</i> manual.
-s	Strips the executable. Causes the 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. See the <code>ld(1)</code> description in the <i>HP-UX Reference</i> manual for more details.
-S	Compiles the named HP C program and leaves the assembly language output in a corresponding file with <i>.s</i> as the suffix.

Option	What It Does
-t <i>x,name</i>	<p>Substitutes or inserts subprocess <i>x</i> with <i>name</i>, where <i>x</i> is one or more of a set of identifiers indicating the subprocesses. This option works in two modes: 1) if <i>x</i> is a single identifier, <i>name</i> represents the full pathname of the new subprocess; 2) if <i>x</i> is a set of identifiers, <i>name</i> represents a prefix to which the standard suffixes are concatenated to construct the full pathnames of the new subprocesses. <i>x</i> can be one or more of the following values:</p> <ul style="list-style-type: none"> • p—Preprocessor (standard suffix is <code>cpp</code>). • c—Compiler (standard suffix is <code>ccom</code>). • 0—Same as <code>c</code>. • a—Assembler (standard suffix is <code>as</code>). • l—Linker (standard suffix is <code>ldr</code>).
-U <i>name</i>	<p>Undefines or removes any initial definition of <i>name</i> in the preprocessor. See the <code>cpp(1)</code> description in the <i>HP-UX Reference manual</i> for details.</p>
-v	<p>Enables the verbose mode sending a step-by-step description of the compilation process to standard error (<code>stderr</code>.)</p>
-V	<p>Prints version information on each invoked subprocess to standard error (<code>stderr</code>.)</p>
-w	<p>Suppresses warning messages.</p>
-W <i>x</i> , <i>arg1[,arg2,...,argn]</i>	<p>Passes the arguments <i>arg1</i> through <i>argn</i> to the subprocess <i>x</i> of the compilation. <i>x</i> can be one of the values described under the <code>-t</code> option with the addition of <code>d</code>, to pass an option to the <code>cc</code> driver. These options are HP value added extensions to HP C.</p>
-W <i>c</i> , -e or +e	<p>Enables the following HP value added features while compiling in ANSI C mode: <code>sized enum</code>, <code>long long</code>, <code>long pointers</code>, compiler supplied defaults for missing arguments to intrinsic calls, and <code>\$</code> in identifier HP C extensions.</p>

Option	What It Does
-W c, -L or +L	<p>Enables any #pragma listing directives and the listing facility. A straight listing prints the following:</p> <ul style="list-style-type: none"> • A banner on the top of each page. • Line numbers. • The nesting level of each statement or declaration. • The postprocessed source file with expanded macros, include files, and no user comments (unless the -C option is used). <p>Under ANSI mode, -Wc, -L causes the compiler to generate an output list of the source without the cpp macro replacements. The -Wc, -Lp option causes the compiler to generate an output list of the source with the cpp macro replacements.</p>
-W c, -m or +m	<p>Causes the identifier maps to be printed. First, all global identifiers are listed, then all the other identifiers are listed by function at the end of the listing. For struct and union members, the address column contains B@b, where B is the byte offset and b is the bit offset. Both B and b are in hexadecimal.</p>
-W c, -o or +o	<p>Causes the code offsets to be printed in hexadecimal grouped by function at the end of the listing.</p>
-W c, or -Rnum	<p>Allow only the first num register variables to actually have the register class. Use this option when the register allocator issues an <i>out of general registers</i> message.</p>
-W c, -wn or +wn	<p>Specifies the level of the warnings messages. The value of n can be one of the following:</p> <ul style="list-style-type: none"> • 1—All warnings are issued. This includes low level warnings that may not indicate anything wrong with the program. • 2—Only warnings indicating that code generation might be affected are issued. This is equivalent to the compiler default without the -w options. • 3—No warnings are issued. This is equivalent to the -w option.

Option	What It Does
-W d, -a	<p>When processing files written in assembly language, this option specifies that the compiler should not assemble with the prefix file that sets up the space and subspace structure required by the linker. Programs assembled with this option may not link unless they contain the equivalent information.</p> <p>You can then optimize the program based on this profile data by re-linking with the +P command line option. The +I command line option is incompatible with the -g, -p, -s, -S, and -y options. The +I option does not affect the default optimization level or the optimization level specified by the -O or +O options. For more details on invoking profile-based optimization, refer to the <i>HP C/HP-UX Programmer's Guide</i>.</p>
-y	<p>Requests the compiler to generate additional information for the static analysis tool, which is a part of the <i>HP SoftBench</i> software development environment. See <i>HP Softbench Static Analyzer: Analyzing Program Structure</i> for more information.</p>
-Y	<p>Enables Native Language Support (NLS) of 8-bit and 16-bit characters in comments, string literals, and character constants. See <code>hpnls(5)</code>, <code>long(5)</code>, and <code>environ(5)</code> in the <i>HP-UX Reference</i> manual for a description of the NLS model. The language value is used to initialize the correct tables for interpreting comments, string literals, and character constants. It is also used to build the pathname to the proper message catalog. Refer to "Location of Files" on page 270 for a description of this path.</p>
-z	<p>Disallows run-time dereferencing of null pointers. Fatal errors result if null pointers are dereferenced.</p>
-Z	<p>Allows dereferencing of null pointers at run-time. (This behavior is the default.) The value of a dereferenced null pointer is zero.</p>

Option	What It Does
+DA <code>model</code>	<p>Generates object code for a particular version of the PA-RISC architecture.</p> <p>If you do not specify this option, the <i>default</i> object code generated is determined automatically as that of the system on which you compile. +DA also specifies which version of the HP-UX math library to link in when you specify -lm. (See the <i>HP-UX Floating-Point Guide</i> for more information about using math libraries.)</p> <p><code>model</code> can be a model number of an HP 9000 system (such as, 730, 877, H50, or I50), one of the PA-RISC architecture designations 1.1, 2.0, 2.0W, or portable. For example, specifying +DA1.1 or +DA735 generates code for the PA-RISC 1.1 architecture. Similarly, specifying +DA2.0 generates 32-bit code for the PA-RISC 2.0 architecture. Specifying +DA2.0W generates 64-bit ELF object files for the PA-RISC 2.0 architecture. Specifying +DAportable generates code compatible across all HP 9000 workstations and servers.</p> <p>See the file /opt/langtools/lib/sched.models for model numbers and their architectures. Use the command <code>uname -m</code> to determine the model number of your system.</p> <p>Note: Object code generated for PA-RISC 2.0 will <i>not</i> execute on PA-RISC 1.1 systems. Object code generated for the 64-bit data model (LP64) will not execute under the 32-bit data model (ILP32). To generate code compatible across PA-RISC 1.1 and 2.0 workstations and servers, use the +DAportable option. For best performance, use +DA with the model number or architecture where you plan to execute the program. Starting at the HP-UX 10.20 release, the default object code generated by HP compilers is determined automatically as that of the machine on which you compile. (Previously, the default code generation was PA-RISC 1.0 on Series 800 servers, and PA-RISC 1.1 on Series 700 workstations.)</p> <p>For more information on this option, see “Using +DA to Generate Code for a Specific Version of PA-RISC” on page 235.</p> <p>See also the +DD32 and +DD64 options.</p>

Option	What It Does
+DCapptype	<p>Generate code for portable or embedded applications (64-bit mode only.) When <i>apptype</i> is <code>hpux</code>, the compiler builds portable applications. This is the default behavior, and is equivalent to using <code>+Oextern</code>. When <i>apptype</i> is <code>emb</code>, the options most commonly used to generate embedded systems code are enabled. Note that code generated with the <code>+DCemb</code> must not be included in a shared library.</p>
+DD32	<p>Generates 32-bit object code for the PA-RISC 1.1 architecture, which runs on PA1.1 and PA2.0 architectures. This option is the same as the options <code>+DA1.1</code> and <code>+DAportable</code>.</p>
+DD64	<p>Generates 64-bit object code for PA2.0 architecture. Causes the macros <code>__LP64__</code> and <code>__PA_RISC2_0</code> to be <code>#defined</code>. This is the same as <code>+DA2.0W</code>, but is the recommended option to use when compiling in 64-bit mode on the PA RISC 2.0 architecture because it is forward compatible with future 64-bit architectures.</p>
+df name	<p>Specifies the path name of the profile database to use with profile-based optimization. This option can be used with the <code>+P</code> command line option. The profile database by default is named <code>flow.data</code>. This file stores profile information for one or more executables. Use <code>+df</code> when the <code>flow.data</code> file has been renamed or is in a different directory than where you are linking.</p> <p>You can also use the <code>FLOW_DATA</code> environment variable to specify a different path and file name for the profile database file. The <code>+df name</code> command line option takes precedence over the <code>FLOW_DATA</code> environment variable.</p> <p>Note, the <code>+df name</code> option cannot be used to redirect the instrumentation output (with the <code>+I</code> option). It is only compatible with the <code>+P</code> option.</p>

Option	What It Does
+DO <i>osname</i>	<p>Sets the target operating system for the compiler. The <i>osname</i> can be:</p> <ul style="list-style-type: none">• 11.0EP9806 (indicates the HP-UX 11.0 EXTPAK 9806 release)• 11.0 (the default) <p>If +DO11.0EP9806 and +Olibcalls are both specified on an HP-UX 11.0EP9806 system, the compiler enables the fusing of math library calls where applicable. This promotes instruction level parallelism by computing the same function (such as <code>sin()</code>) of two values concurrently.</p> <p>+DO can be used at any optimization level. By default, when you compile your application, it is binary compatible across the 11.x release. You only need to specify +DO when you want the latest performance features supported in the OS.</p> <p><i>Caution:</i> Use of +DO with a non-default <i>osname</i> makes the resulting code binary incompatible with an earlier version of HP-UX.</p>

Option	What It Does
+DS <i>model</i>	<p>Performs instruction scheduling tuned for a particular implementation of the PA-RISC architecture.</p> <p><i>model</i> can be a model number of an HP 9000 system (such as 730, 877, H50, or I50); a PA-RISC architecture designation 1.1 or 2.0; or a processor name (such as PA7100LC). For example, specifying +DS720 performs instruction scheduling tuned for one implementation of PA-RISC 1.1. Specifying +DSPA7100LC performs scheduling for a system with a PA7100LC processor. Specifying +DSPA8000 performs instruction scheduling for systems based on the PA-RISC 8000 processor.</p> <p>To obtain the best performance on a particular model or processor of the HP 9000, use +DS with that model number or processor name.</p> <p>If you plan to run your program on both PA-RISC 1.1 and 2.0 systems, in general we recommend you use +DS2.0.</p> <p>See the file /opt/langtools/lib/sched.models for model numbers and the processor names for which implementation-specific scheduling is performed. Use the command <code>uname -m</code> to determine the model number of your system.</p> <p>Object code with scheduling tuned for a particular model or processor <i>will</i> execute on other HP 9000 systems, although possibly less efficiently.</p> <p>For more information on this option, see “Using +DS to Specify Instruction Scheduling” on page 236.</p>
+e	See -W c, -e
+ESconstlit	<p>Introduces a new default behavior for the HP C compiler, in which HP C stores constant-qualified (<code>const</code>) objects and literals in read-only memory. Storing <code>const</code>-qualified string literals in read-only memory can cause a program which violates the semantics of <code>const</code> to abnormally terminate with a bus error and core dump. This is because literals, which have been placed in read-only memory, may not be modified at runtime. See the description of +ESnolite for details on how to specify that literals not be placed in read-only memory.</p>

Option	What It Does
+ESfic	<p>The +ESfic option generates code for fast indirect calls. This option should only be used in an environment where there are no dependencies on shared libraries. The application must be linked with archive libraries. Using this option can improve run-time performance.</p>
+ESlit	<p>Places string literals and constants defined with the ANSI C <code>const</code> type qualifier into the <code>\$LIT\$</code> subspace. The <code>\$LIT\$</code> subspace is used for read-only data storage. This option can reduce memory requirements and improve run-time speed in multi-user applications.</p> <p>Normally the C compiler only places floating-point constant values in the <code>\$LIT\$</code> subspace, and other constants and literals in the <code>\$DATA\$</code> subspace. This option allows the placement of large data objects, such as ANSI C <code>const</code> arrays, into the <code>\$LIT\$</code> subspace.</p> <p>All users of an application share the static data stored in the <code>\$LIT\$</code> subspace. Each user is allocated a private copy of the dynamic data stored in the <code>\$DATA\$</code> subspace. By moving additional static data from the <code>\$DATA\$</code> subspace to the <code>\$LIT\$</code> subspace, overall system memory requirements can be reduced and run-time speed improved. Most applications can benefit from this option.</p> <p>Users should not attempt to modify string literals if they use the +ESlit option. The reason is that this option places all string literals into read-only memory. Particularly, the following C library functions should be used with care, since they can alter the contents of string literals if users specify string literals as the receiving string.</p> <pre>extern char *strncat(char *, const char *, size_t); extern void *memmove(void *, const void *, size_t); extern char *strcpy(char *, const char *); extern char *strncpy(char *, const char *, size_t); extern char *strcat(char *, const char *); extern char *strtok(char *, const char *);</pre>

Option	What It Does
+ESnolit	<p>+ESnolit disables the default behavior of the HP C compiler to store literals in read-only memory. Specifying this option causes HP C to no longer store literals in read-only memory. This restores HP C's traditional behavior prior to this release of the compiler.</p>
+ESnoparmreloc	<p>+ESnoparmreloc disables parameter relocation for function calls. If your source code uses the ANSI C function prototype consistently in both declaration and definition of a function, this option may allow you to produce smaller and faster function entry code.</p> <p>The library is compiled with this option off. So, to use this option in your code, you must declare the library functions without function prototype.</p> <p>For example, <code>cos</code> has not been compiled with +ESnoparmreloc. If you declare <code>cos</code> with a function prototype as shown below, and compile with +ESnoparmreloc, an incorrect value will be passed to <code>cos()</code>.</p> <pre>double cos(double); main() { printf("%f\n", cos(1.0)); }</pre> <p>For more information on parameter relocation, see the <i>HP PA-RISC Procedure Calling Conventions Reference Manual</i>, HP part number 09740-90015.</p>

Option	What It Does
<code>+ESsfc</code>	<p>Replaces millicode calls with inline code when performing simple function pointer comparisons. The <code>+ESsfc</code> compiler option affects how function pointers are compared in generated code. The default is to generate low-level millicode calls for function pointer comparisons.</p> <p>The <code>+ESsfc</code> option generates code that compares function pointers directly, as if they were simple integers.</p> <p>The <code>+ESsfc</code> option should only be used in an environment where there are no dependencies on shared libraries. The application must be linked with archive libraries. Using this option can improve run-time performance.</p>
<code>+f</code>	<p>Inhibits the automatic promotion of <code>float</code> to <code>double</code> when evaluating expressions. This differs from the <code>+r</code> option in that both parameters <i>and</i> function return values are still promoted to <code>double</code>. In ANSI mode, this option is ignored and a warning is issued.</p>

Option	What It Does																				
+FP <i>flags</i>	<p>Specifies what floating-point traps to enable and also enables or disables fast underflow mode. <i>flags</i> is a series of upper case or lower case letters from the set [VvZzOoUuIiDd] with no spaces, tabs, or other characters between them. If the upper-case letter is selected, that behavior is enabled. If the lower-case letter is selected or if the letter is not present in the flags, the behavior is disabled. By default, all traps are disabled. The values for <i>flags</i> are:</p> <table border="0"> <tr> <td data-bbox="597 758 613 779">V</td> <td data-bbox="808 758 1276 821">Enable traps on invalid floating-point operations.</td> </tr> <tr> <td data-bbox="597 842 613 863">v</td> <td data-bbox="808 842 1276 905">Disable traps on invalid floating-point operations.</td> </tr> <tr> <td data-bbox="597 926 613 947">Z</td> <td data-bbox="808 926 1393 1010">Enable traps on divide by zero. (If your program must conform to the POSIX standard, do not enable this trap.)</td> </tr> <tr> <td data-bbox="597 1031 613 1052">z</td> <td data-bbox="808 1031 1195 1062">Disable traps on divide by zero.</td> </tr> <tr> <td data-bbox="597 1083 613 1104">O</td> <td data-bbox="808 1083 1292 1115">Enable traps on floating-point overflow.</td> </tr> <tr> <td data-bbox="597 1136 613 1157">o</td> <td data-bbox="808 1136 1292 1167">Disable traps on floating-point overflow.</td> </tr> <tr> <td data-bbox="597 1178 613 1199">U</td> <td data-bbox="808 1178 1308 1209">Enable traps on floating-point underflow.</td> </tr> <tr> <td data-bbox="597 1230 613 1251">u</td> <td data-bbox="808 1230 1308 1262">Disable traps on floating-point underflow.</td> </tr> <tr> <td data-bbox="597 1283 613 1304">I</td> <td data-bbox="808 1283 1373 1346">Enable traps on floating-point operations that produce inexact results.</td> </tr> <tr> <td data-bbox="597 1367 613 1388">i</td> <td data-bbox="808 1367 1373 1430">Disable traps on floating-point operations that produce inexact results.</td> </tr> </table>	V	Enable traps on invalid floating-point operations.	v	Disable traps on invalid floating-point operations.	Z	Enable traps on divide by zero. (If your program must conform to the POSIX standard, do not enable this trap.)	z	Disable traps on divide by zero.	O	Enable traps on floating-point overflow.	o	Disable traps on floating-point overflow.	U	Enable traps on floating-point underflow.	u	Disable traps on floating-point underflow.	I	Enable traps on floating-point operations that produce inexact results.	i	Disable traps on floating-point operations that produce inexact results.
V	Enable traps on invalid floating-point operations.																				
v	Disable traps on invalid floating-point operations.																				
Z	Enable traps on divide by zero. (If your program must conform to the POSIX standard, do not enable this trap.)																				
z	Disable traps on divide by zero.																				
O	Enable traps on floating-point overflow.																				
o	Disable traps on floating-point overflow.																				
U	Enable traps on floating-point underflow.																				
u	Disable traps on floating-point underflow.																				
I	Enable traps on floating-point operations that produce inexact results.																				
i	Disable traps on floating-point operations that produce inexact results.																				

Compiling and Running HP C Programs
Compiling HP C Programs

Option	What It Does
	<p>D Enable fast underflow (flush to zero) of denormalized values. (Enabling fast underflow is an undefined operation on PA-RISC 1.0 based systems, but it is defined on all subsequent versions of PA-RISC. Selecting this value enables fast underflow only if it is available on the processor that is used at run time.)</p> <p>d Disable fast underflow (flush to zero) of denormalized values.</p> <p>To dynamically change these settings at run time, refer to <i>fpgetround(3M)</i>.</p>
+help	Launches a web browser displaying an html version of the HP C/HP-UX online help, and then processes any other arguments. See “HP C Online Help” on page 5 for instructions on setting up your environment to view the HP C Online Help.

Option	What It Does
<code>+hugesize=<i>n</i>[M][K]</code>	<p>Lowers the threshold for huge data.</p> <p><i>n</i> is an integer with a maximum value of 2^{58} bytes.</p> <p>M indicates megabytes.</p> <p>K indicates kilobytes.</p> <p>M or K are optional specifiers. If neither is specified the default is K.</p> <p>In order for the compiler to qualify a data object as huge, its size must exceed the current setting of the <code>hugesize</code> threshold. The threshold is initially set to 256 megabytes (2^{28} bytes). A data object whose size exceeds the threshold is allocated in huge subspace.</p> <p>The compiler puts all small arrays, structures, and unions into a separate subspace. An application with too many arrays may run out of subspace creating an overflow condition. In this case linking fails. One way to solve this problem is to lower the threshold so that any array greater than or equal to the threshold (in bytes) is put into the huge data subspace instead.</p> <p>The following example lowers the threshold to 2^{23} (8 megabytes) resulting in more data objects being put in huge subspace:</p> <pre>cc +DD64 +hugesize=8M appl.c app2.c</pre> <p>For more information, see <code>cc +help</code>.</p>
<code>+I</code>	<p>Instructs the compiler to instrument the object code for collecting run-time profile data. The profiling information can then be used by the linker to perform profile-based optimization. Code generation and optimization phases are delayed until link time by this option.</p> <p>After compiling and linking with <code>+I</code>, run the resultant program using representative input data to collect execution profile data. Profile data is stored in <code>flow.data</code> by default. See the <code>+dfname</code> option for information on controlling the name and location of this data file.</p>

Option	What It Does
+k	<p>Generates long-displacement code sequences so a program can reference large amounts of global data that is physically located in shared libraries. Only use +k when you get a linker message indicating you need to use it.</p> <p>By default, the HP C compiler generates short-displacement code sequences for programs that reference global data that is physically located in shared libraries. For nearly all programs, this is sufficient.</p> <p>If your program references a large amount of global data in shared libraries, the default code generation for referencing that global data may not be sufficient. If this is the case, when you link your program the linker will give an error message indicating that you need to recompile your program with the +k option.</p>
+L	See -W c, -L
+m	See -W c, -m
+M0 (or +M)	<p>Emits ANSI C migration warnings. Use the +M option with either -Aa or -Ac. These were formerly part of +w1.</p> <p>The document <i>ANSI Programming Language C Standard ISO 9899:1990</i> discusses the behavioral differences in code that is migrated to ANSI C. These differences are called quiet changes.</p>
+M1	Emit warnings for possible problems connected with platform migration.
+M2	Emit warnings for possible problems connected with migration to the LP64 data model (64-bit mode). This option is only available in 64-bit mode. Refer to the <i>HP-UX 64-bit Porting and Transition Guide</i> for details.
+o	See -W c, -o

Option	What It Does								
+[no]objdebug	<p>When used with -g, +objdebug leaves debug information in the object files instead of copying it to the executable file at link time, resulting in shorter link times and smaller executables. The default, +noobjdebug, copies the debug information to the executable file.</p> <p>When you specify -g, the compiler places symbolic debugging information into the object files. By default, the linker calls pxdb which compacts this debug information & copies it to the executable file. When +objdebug was used at compile time, the linker leaves the debug information in the object files. To debug the executable file, the HP WDB debugger must have access to the object files. If you move the object files, use HP WDB's objdir command to tell it where the object files are. (The HP DDE debugger does not support this option.) This option reduces link time and the size of the executable file by avoiding this copying of debug information.</p> <p>The compile-time default is +noobjdebug. If the linker detects any object files that were compiled with +objdebug, it will leave the debug information in those files. Any object files not compiled with +objdebug will have their debug information copied into the executable file. You can leave debug information in some object files and not in others.</p> <p>Use the +noobjdebug option when linking to explicitly tell the linker to copy all debug information to the executable file, even from files compiled with +objdebug.</p>								
+Oopt	<p>Invokes the level of optimization selected by <i>opt</i>. <i>opt</i> can have any of the following values.</p> <table border="0"> <tr> <td style="padding-right: 20px;">1</td> <td>Performs level 1 optimizations.</td> </tr> <tr> <td>2</td> <td>Performs level 1 and 2 optimizations.</td> </tr> <tr> <td>3</td> <td>Performs level 1, 2, and 3 optimizations.</td> </tr> <tr> <td>4</td> <td>Performs level 1, 2, 3, and 4 optimizations.</td> </tr> </table> <p>For a complete list of advanced optimization toggles, see Chapter 4, "Optimizing HP C Programs," in the <i>HP C/HP-UX Programmer's Guide</i>.</p>	1	Performs level 1 optimizations.	2	Performs level 1 and 2 optimizations.	3	Performs level 1, 2, and 3 optimizations.	4	Performs level 1, 2, 3, and 4 optimizations.
1	Performs level 1 optimizations.								
2	Performs level 1 and 2 optimizations.								
3	Performs level 1, 2, and 3 optimizations.								
4	Performs level 1, 2, 3, and 4 optimizations.								

Option	What It Does
+P	<p>Directs the compiler to use profile information to guide code generation and profile-based optimization. This option causes the compiler to generate intermediate compiler code instead of compiled object code. The actual code generation is done at link time.</p> <p>The +P option does not affect the default optimization level, or the optimization level specified by the -O or +O<i>opt</i> options. Note: Source files that are compiled with the +I option do not need to be recompiled with +P in order to use profile-based optimization. You only need to relink the object files with the +P option after running the instrumented version of the program.</p> <p>The +P command line option is incompatible with the +I, -g, -s, -S, and -y options.</p> <p>Also refer to the +I, +pgm and +df command line options. For more information on using these options, see the <i>HP C/HP-UX Programmer's Guide</i>.</p>
+pa	<p>Requests that the application be compiled for routine-level profiling with CXperf. The +pa option is not valid with the +O4 or +Oall optimization options. Also, +pa is not compatible with the -p or -G options. For more information, see the <i>CXperf Online Help</i> or <i>CXperf Command Reference</i>.</p>
+pal	<p>Requests that the application be compiled for routine-level and loop-level profiling with CXperf. The +pal option is not valid with the +O4 or +Oall optimization options. Also, +pal is not compatible with the -p or -G options. For more information, see the <i>CXperf User's Guide</i> on http://www.docs.hp.com.</p>
+pgm <i>name</i>	<p>Specifies a program name to look up in the flow.data file. Used with profile-based optimization and the +P option.</p> <p>+pgm<i>name</i> should be used when the name of the profiled executable differs from the name of the current executable specified by the -o option.</p>
+r	<p>Inhibits the automatic promotion of float to double when evaluating expressions and passing arguments. This option is not valid in ANSI mode.</p>

Option	What It Does
+sb	Use signed bitfields (in 32- and 64-bit modes). By default, unqualified bitfield types are signed in 32-bit mode and unsigned in 64-bit mode.
+u <code>bytes</code>	Forces all pointer references to assume that data is aligned on either an 8-bit, 16-bit, or 32-bit addresses. <code>bytes</code> can be 1 (8-bit), 2 (16-bit), or 4 (32-bit). The default value for +u is 2. This option can be used when reading in non-natively aligned data. Pragmas are also available to assist with processing non-natively aligned data. See Chapter 2, “Storage and Alignment Comparisons,” in the <i>HP C/HP-UX Programmers Guide</i> for more information.
+uc	Makes unqualified <code>char</code> data types unsigned. By default, the HP C compiler makes all unqualified <code>char</code> data types signed. Use this option to help port applications from other vendor platforms, where the default behavior for unqualified <code>char</code> types is unsigned. Be careful when using this option. Your application may have trouble interfacing with HP-UX system libraries and other libraries that do not use this option.
+wn	<p>Specifies the level of the warnings messages. The value of <code>n</code> can be one of the following:</p> <ul style="list-style-type: none"> • 1—All warnings are issued. This includes low level warnings that may not indicate anything wrong with the program. • 2—Only warnings indicating that code generation might be affected are issued. This is equivalent to the compiler default without the <code>-w</code> options. • 3—No warnings are issued. This is equivalent to the <code>-w</code> option. <p>This option is the same as <code>-W c, -wn</code>.</p>
+W <code>n1[,n2[,...nN]]</code>	Suppresses the specified warnings, where <code>n1</code> through <code>nN</code> are valid compiler warning message numbers. See the file <code>/opt/ansic/lib/nls/msg/C/cc.msgs</code> for a list of error and warning messages.
+We <code>n1[,n2[,...nN]]</code>	Changes the specified warning to an error, where <code>n1</code> through <code>nN</code> are valid compiler warning messages. See the file <code>/opt/ansic/lib/nls/msg/C/cc.msgs</code> for a list of compiler messages.

Compiling and Running HP C Programs
 Compiling HP C Programs

Option	What It Does
+Ww <i>n1</i> [, <i>n2</i> [,... <i>nN</i>]]	Enables the specified warnings, assuming all other warnings have been suppressed with -w or +w3. <i>n1</i> through <i>nN</i> are valid compiler warning messages. See the file /opt/ansic/lib/nls/msg/C/cc.msgs for a list of compiler messages.
+z	Generates object code that can be added to a shared library. Object code generated with this option is position independent, containing no absolute addresses. All addresses are either pc-relative or indirect references. This is the same as the +Z option in 64-bit mode. The -p and -G options are incompatible with this option and are ignored. See the <i>HP-UX Linker and Libraries Online User Guide</i> for detailed information on shared libraries.
+Z	This option is similar to +z with the difference being that space for more imported symbols is allocated. The size of the data linkage table allocated by the +z and +Z options is machine dependent. Use the +Z option when the linker ld issues an error message indicating data linkage table overflow. +Z is the default in 64-bit mode. For more information on shared libraries, see the <i>HP-UX Linker and Libraries Online User Guide</i> .
Any other option encountered generates a warning to standard error (stderr.) (Options not recognized by cc are <i>not</i> passed on to ld. Use the -Wl, arg option to pass options to ld.)	

Examples of Compiler Commands

- `cc -Aa prog.c`
requests a strict ANSI C compilation of `prog.c`.
- `cc -tp,/users/devel/cpp prog.c`
uses `/users/devel/cpp` as the pathname for the preprocessing phase.
- `cc -tpca,/users/devel/x prog.s`
uses `/users/devel/x/cpp` for `cpp`, `/users/devel/x/ccom` for `ccom`, and `/users/devel/x/as` for `as`; the assembly file `prog.s` is processed by the specified assembler.
- `cc -Aa prog.c procedure.o -o prog`
compiles and links the file `prog.c`, creating the executable program file `prog`. The compiler produces `prog.o`. The linker `ld(1)` links `prog.o` and `procedure.o` with all of the HP C startup routines in `/lib/crt0.o` and the library routines from the HP C library `/lib/libc.a`.
- `cc prog.c -co /users/my/prog.o`
compiles the source file `prog.c` and places the object file `prog.o` in `/users/my/prog.o`.
- `cc -Wp,-H150000 p1.c p2.c p3.c -o p`
compiles the source files in the option `-H150000` to the preprocessor `cpp` to increase the define table size from the default.
- `cc -Wl,-vt *.c -o vmh`
compiles all files in the working directory ending with `.c`, passes the `-vt` option to the linker, and causes the resulting program file to be named `vmh`.
- `cc -vg prog.c`
compiles `prog.c`, adds debug information, and displays the steps in the compilation process.
- `cc -S prog.c`
compiles the file `prog.c` into an assembly output file called `prog.s`.

Compiling HP C Programs

- `cc -Wc,-R15 prog.c`
compiles `prog.c` placing no more than the first fifteen declared register variables in each function into registers.
- `cc -Wc,-L,-m,-o prog.c`
compiles `prog.c` and produces a source listing, local and global identifier maps, and local code offsets on the standard output device.
- `cc -OAa prog.c`
compiles `prog.c` in ANSI mode and requests level 2 optimization.
- `cc +O1 prog.c`
compiles `prog.c` and requests level 1 optimization.
- `cc +wl prog.c -c`
compiles `prog.c` with low-level warnings emitted and suppresses linking.
- `cc -D BUFFER_SIZE=1024 prog.c`
passes the option `-D BUFFER_SIZE=1024` to the preprocessor, setting the value of the macro for the compilation of `prog.c`.
- `cc -lm prog.c`
compiles `prog.c` requests the linker to link the library `/lib/libm.a` with the object file `prog.o` to create the executable `a.out`.
- `cc -L/users/devel/lib -lme prog.c`
compiles `prog.c` and causes the linker to search the directory `/users/devel/lib` for the library `libme.a`, before searching in `/lib` or `/usr/lib` for it.
- `cc +DA1.1 +Z -c prog.c`
compiles `prog.c` for use in a large shared library in 32-bit mode.
- `cc +DA2.0W +Z -c prog.c`
compiles `prog.c` for use in a large shared library or executable in 64-bit mode.
- `cc +DA1.1 +z -c prog.c`
compiles `prog.c` in 32-bit mode for use in a shared library with less than 16K of symbols.

- `cc +DA2.0W -noshared +Onoextern prog.c`
compiles `prog.c` in 64-bit mode without shared libraries. (Refer to the discussion on optimization options in the C online help or the *HP C/HP-UX Programmer's Guide* for information on the `+O[no]extern` option.)

Environment Variables

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

- CCOPTS
- TMPDIR

CCOPTS Environment Variable

You can pass arguments to the compiler using the CCOPTS environment variable or include them on the command line. The CCOPTS environment variable provides a convenient way for establishing default values for `cc` command line options. It also provides a way for overriding `cc` command line options.

The syntax for the CCOPTS environment variable in C shell notation is:

```
setenv CCOPTS [options] [ | [options]]
```

The compiler places the arguments that appear before the vertical bar in front of the command line arguments to `cc`. It then places the second group of arguments after any command line arguments to `cc`.

Options that appear after the vertical bar in the CCOPTS variable override and take precedence over options supplied on the `cc` command line.

If the vertical bar is omitted, the compiler gets the value of CCOPTS and places its contents before any arguments on the command line.

For example, the following in C shell notation

```
setenv CCOPTS -v  
cc -g prog.c
```

is equivalent to

```
cc -v -g prog.c
```

Compiling and Running HP C Programs

Compiling HP C Programs

For example, the following in C shell notation

```
setenv CCOPTS "-v | +O1"  
cc +O2 prog.c
```

is equivalent to

```
cc -v +O2 prog.c +O1
```

In the above example, level 1 optimization is performed, since the +O1 argument appearing after the vertical bar in CCOPTS takes precedence over the cc command line arguments.

TMPDIR Environment Variable

Another environment variable, `TMPDIR`, allows you to change the location of temporary files that the compiler creates and uses. The directory specified in `TMPDIR` replaces `/var/tmp` as the default directory for temporary files. The syntax for `TMPDIR` in C shell notation is:

```
setenv TMPDIR altdir
```

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

Compiling for Different Versions of the PA-RISC Architecture

This section discusses the use of the `+DA` and `+DS` options in more detail.

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

By default, compiling on different HP 9000 systems produces code for the architecture of the system on which the compilation is performed. Use the `+DA` option to change this default behavior.

The `+DA` option specifies which PA-RISC instruction set the compiler should use when generating code. Specifying `+DAportable` ensures your code will run on HP PA-RISC 2.0 and 1.1 systems, although the performance of your program may not be as good as it could be if optimized for a specific system. 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. Specifying `+DA2.0` gives optimal performance on PA-RISC 2.0 systems, but the program will not run on the earlier PA-RISC architectures.

Use the command `uname -m` to determine the model number of your system.

When you use the `+DA` option depends on your particular circumstance.

- If you plan to run your program on the same system where you are compiling, you don't need to use `+DA`.
- 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 `+DAmodel` 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`. This will give you the best performance on the 855.

- If you plan to run your program on PA-RISC 2.0 and 1.1 HP 9000 systems, use `+DAportable` to ensure portability.

If you do not specify `+DA` or `+DS`, the default instruction scheduling is based on that of the system on which you compile. If you do specify a `+DA` option and do not specify a `+DS` option, the default instruction scheduling is based on what you specify in `+DA`, and not based on that of the system on which you compile. For example, if you specify `+DA1.1` and do not specify `+DS`, and instruction scheduling will be for 1.1. If you specify `+DAportable` and do not specify `+DS`, and instruction scheduling will be for 1.1. (`+DAportable` is currently equivalent to `+DA1.1`.)

Using `+DS` to Specify Instruction Scheduling

Instruction scheduling is different on different implementations of PA-RISC architectures. You can improve performance on a particular model or processor of the HP 9000 by requesting that the compiler use instruction scheduling tuned to that particular model or processor. Using scheduling for one model or processor does not prevent your program from executing on another model or processor.

Use the `+DS` option to specify instruction scheduling tuned to a particular implementation of PA-RISC. Note that *model* can be a model number of an HP 9000 system (such as 730, 877, or H40); a PA-RISC architecture designation 1.1 or 2.0; or one of the PA-RISC processor names (such as PA7000, PA7100, PA7100LC, or PA8000.)

For example, to specify instruction scheduling for the model 867, use `+DS867`. To specify instruction scheduling for the PA-RISC 7100LC processor, use `+DSPA7100LC`. To specify instruction scheduling for systems based on the PA-RISC 8000 processor, use `+DSPA8000`.

If you plan to run your program on both PA-RISC 1.1 and 2.0 systems, in general we recommend you use the `+DS2.0` designation.

See the file `/opt/langtools/lib/sched.models` for model numbers and processor names. Use the command `uname -m` to determine the model number of your system.

If you do not specify a `+DA` or `+DS` option, the default instruction scheduling is based on that of the system on which you compile. If you do specify a `+DA` option and do not specify a `+DS` option, the default instruction scheduling is based on what you specify in `+DA`, and not based on that of the system on which you compile. For example, if you specify `+DA1.1` and do not specify `+DS`, and instruction scheduling will be for 1.1. Specify `+DAportable` and do not specify `+DS`, and instruction scheduling will be for 1.1. (`+DAportable` is currently equivalent to `+DA1.1`.)

When you use the `+DS` option depends on your particular circumstance.

- 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 `+DSmodel` with either the model number of the target system or the processor name of the target system.

For example, if you are compiling on a system with a PA7100 processor and your program will run on a system with a PA7100LC processor, you should use `+DSPA7100LC`. This will give you the best performance on the PA7100LC system.

- If you plan to run your program on many models or processors of the HP 9000, use `+DSmodel` with either the model number or processor name of the fastest system on which you will be running 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

A `#pragma` directive is an instruction to the compiler. Put pragmas in your C source code where you want them to take effect, but do not use them within a function. A pragma has effect from the point at which it is included to the end of the translation unit (or until another pragma changes its status).

This section introduces the following groups of HP C compiler directives:

- initialization and termination pragmas
- intrinsic pragmas
- copyright notice and identification pragmas
- data alignment pragmas
- optimization pragmas
- program listing pragmas
- shared library pragmas
- fast libcall pragmas

Refer to the *HP C/HP-UX Programmer's Guide* for information on parallel processing pragmas and for additional information.

Initialization and Termination Pragmas

This section describes the `INIT` and `FINI` pragmas. These allow the user to set up functions which are called when a load module (a shared library or executable) is loaded (*initializer*) or unloaded (*terminator*). For example, when a program begins execution, its initializers get called before any other user code gets called. This allows some set up work to take place. In addition, when the user's program ends, the terminators can do some clean up. When a shared library is loaded or unloaded with the `shl_load` or `dlopen` API, its initializers and terminators are also executed at the appropriate time.

INIT Pragma

```
#pragma INIT "string"
```

Use the compiler pragma `INIT` to specify an initialization function. The functions take no arguments and return nothing. The function specified by the `INIT` pragma is called before the program starts or when a shared library is loaded.

For example:

```
#pragma INIT "my_init"
void my_init() { ... do some initializations ... }
```

FINI Pragma

```
#pragma FINI "string"
```

Use the compiler pragma `FINI` to specify a termination function. The function specified by the `FINI` pragma is called after the C program terminates by either calling the `libc` `exit()` function, returning from the `main` or `_start` functions, or when the shared library which contains the `FINI` is unloaded from memory. Like the function called by the `INIT` pragma, the termination function takes no arguments and returns nothing.

For example:

```
#pragma FINI "my_fini"
void my_fini() { ... do some clean up ... }
```

The Linker +I Option

HP-UX 10.x style initializers are the same type supported in all HP-UX 10.x releases. These are called both before the user's code is started or a shared library is loaded as well as when the shared library is unloaded. The linker option `+I` is used to create this type of initializer.

Compiling and Running HP C Programs

Pragmas

The function returns nothing but takes two arguments. The first is a handle to the shared library being initialized. This handle can be used in calling *shl_load* API routines. The second is set to non-zero at startup and zero at program termination.

```
$ ld -b foo.o +I my_10x_init -o libfoo.sl
#include <dl.h>
void my_10x_init(shl_t handle, int loading)
{
/* handle is the shl_load API handle for the shared library being
initialized. */
/* loading is non-zero at startup and zero at termination. */
if (loading) {
... do some initializations ...
} else {
... do some clean up ...
}
}
```

This type is only supported when creating a shared library (linker *-b* option). Also note that it may not be called when the user's program terminates. They are always called, however, if the library is explicitly unloaded as a result of a *shl_load* or *dlopen* API call.

All three types (INIT, FINI, and +I) are supported in 64-bit links. Only HP-UX 10.x style initializers are supported in 32-bit links.

Intrinsic Pragmas

See Chapter 11, "Using Intrinsic," on page 275 for further information about the pragmas introduced here.

INTRINSIC Pragma

```
#pragma INTRINSIC intrinsic_name1 [user_name] [intrinsic_name2]
[user_name]...
```

Declares an external name as an intrinsic.

INTRINSIC_FILE Pragma

```
#pragma INTRINSIC_FILE "path"
```

Specifies the path of a file in which the compiler can locate information about intrinsic functions.

See Chapter 11, “Using Intrinsic,” on page 275 for further information about the intrinsic pragmas.

Copyright Notice and Identification Pragmas

The following pragmas can be used to insert strings in code.

COPYRIGHT Pragma

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

Places a copyright notice in the object file, using the “*string*” argument and the date specified using `COPYRIGHT_DATE`. If no date has been specified using `#pragma COPYRIGHT_DATE`, 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.
```

COPYRIGHT_DATE Pragma

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

Specifies a date string to be used in a copyright notice appearing in an object module.

LOCALITY Pragma

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

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"` will build the name `"$CODE$MINE$"`.

Code that is not headed by a `LOCALITY` pragma is associated with the name `$CODE$`. An empty “*string*” causes the code name to revert to the default name of `$CODE$`.

VERSIONID Pragma

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

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.

Data Alignment Pragmas

This section discusses the data alignment pragmas `HP_ALIGN` and `PACK` and their various arguments available on the HP 9000 workstations and servers, to control alignment across platforms. In the following discussion, a word represents a 32-bit data structure. Refer to Chapter 2, “Storage and Alignment Comparisons,” in the *HP C/HP-UX Programmer's Guide* for detailed information on the `HP_ALIGN` and `PACK` pragmas.

HP_ALIGN Pragma

```
#pragma HP_ALIGN align_mode [PUSH]
```

```
#pragma HP_ALIGN POP
```

Data Alignment Stack . The `PUSH` and `POP` options allow functions to establish their own alignment environment for the duration of the function call, and restore the alignment environment previously in effect when exiting the procedure.

The `HP_ALIGN` pragma must have a global scope (outside of any function, enclosing structure, or union).

```
#pragma HP_ALIGN POP
```

This option to `HP_ALIGN` restores the previous alignment environment saved using the `HP_ALIGN PUSH` pragma by deleting the current alignment mode from the top of the stack. If the alignment stack is empty, the default alignment is made the current mode.

```
#pragma HP_ALIGN align_mode PUSH
```

This pragma saves the current alignment environment for later retrieval by pushing it into a last in, first out (LIFO) stack. The *align_mode* is made the new current alignment by placing it on the top of the stack.

Alignment Modes . *align_mode* can be any of the following values:

```
#pragma HP_ALIGN HPUX_WORD
```

Results in `int` and `float` types being halfword aligned (two-byte aligned), `doubles` being word aligned (four byte aligned), and all `structs` being at least halfword aligned. This is the default for the Series 300/400 computer.

```
#pragma HP_ALIGN HPUX_NATURAL_S500
```

Results in `doubles` being word aligned. This is the default for the Series 500 computer.

```
#pragma HP_ALIGN HPUX_NATURAL
```

Results in native alignment for the HP 9000 workstations and servers. The `int` and `float` types are word aligned, `doubles` are double-word aligned (8-byte aligned), and `structs` may be byte aligned depending upon the data types used within the structure.

```
#pragma HP_ALIGN NATURAL
```

Results in a superset of the above. Uses native alignment provided by `HPUX_NATURAL`, all `structs` and `unions` are at least halfword aligned, and `DOMAIN` bit-field mapping is used. (See Chapter 2, "Storage and Alignment Comparisons," in the *HP C/HP-UX Programmer's Guide* for details regarding Domain bit-fields.)

```
#pragma HP_ALIGN DOMAIN_NATURAL
```

Similar to `NATURAL` except `long doubles` are only 8 bytes long (treated as `doubles`).

```
#pragma HP_ALIGN DOMAIN_WORD
```

Similar to `HPUX_WORD`, with the following exceptions: `long doubles` are only 8 bytes long (treated as `doubles`) and `DOMAIN` bit-field mapping is used. (See Chapter 2, "Storage and Alignment Comparisons," in the *HP C/HP-UX Programmer's Guide* for details regarding Domain bit-fields.)

```
#pragma HP_ALIGN NOPADDING
```

Causes all structure and union members that are not bit-fields to be packed on a byte boundary. It does *not* cause crunched data packing, where there are zero bits of padding. It only ensures that there will be no full bytes of padding in the structure or union.

Accessing Data with the HP_ALIGN Pragma . The HP_ALIGN pragma isolates data structures that are not naturally aligned for PA-RISC systems.

References to non-natively aligned data often results in poorer run-time performance than references to natively aligned data. Natively aligned data is often accessed with a single load or store instruction.

Non-natively aligned data must be accessed with one or more load and store instructions.

The HP_ALIGN pragma differs from the *+ubytes* compiler option. When you use the HP_ALIGN pragma, the compiler localizes inefficient code generation to accesses of data declared with structures or unions under the HP_ALIGN pragma. The *+ubytes* option assumes that all references to pointer objects are misaligned and performs worst case code generation for all loads and stores of dereferenced data.

The HP_ALIGN pragma offers better run-time performance than the *+unumber* option. However, the HP_ALIGN pragma requires careful pointer assignment and dereferencing. The following example shows how the pragma is often misused, and what can be done to correct the mistake.

```
#pragma HP_ALIGN HPUX_WORD

struct {
    char chardata;
    int intdata;
} stvar;

main()
{
    int *localptr;
    int localint;
    localptr = &stvar.intdata;
    localint = *localptr; /* invalid dereference */
}
```

The above program aborts at run-time when *localptr* is dereferenced. The structure *stvar* is declared under the HP_ALIGN HPUX_WORD pragma. Its members will not be natively aligned for PA-RISC. The member *stvar.intdata* is aligned on a two byte boundary.

The error occurs after the address of *stvar.intdata* is assigned to *localptr*. *localptr* is not affected by the HP_ALIGN HPUX_WORD pragma. When *localptr* is used to access the data, it is treated as a pointer to four-byte aligned data rather than as a pointer to two-byte aligned data and a run-time error can occur.

Two solutions help to work around this problem. First, the recommended solution is to access non-natively aligned data through structures or unions that are declared under the `HP_ALIGN` pragma.

For example, the above program can be transformed to:

```
#pragma HP_ALIGN HPUX_WORD

struct {
    char chardata;
    int intdata;
} stvar; /* stvar is declared under the HP_ALIGN pragma */

main()
{
    int *localptr;
    int localint;
    localint = stvar.intdata; /* Valid access of non-naturally */
                             /* aligned data through a struct */
}
```

The second method is to inform the compiler that all access of data must assume potentially non-natural alignment. In the case of `#pragma HP_ALIGN HPUX_WORD` shown in the first example above, you can use the `+u2` command line option. This causes all loads and stores of any data to be performed in increments of no greater than 2-bytes at a time.

For complete details about the `+ubytes` option, see Table 9-2 on page 207.

PACK Pragma

```
#pragma PACK n
```

The `PACK` pragma is a simple, intuitive way for users to specify alignment. In the syntax, *n* is the byte boundary on which members of structs and unions should be aligned, and can be 1, 2, 4, 8, or 16.

The `PACK` pragma is not intended to be an “extension” of the `HP_ALIGN` pragma. It is, instead, a simple and highly portable way of controlling the alignment of aggregates. It has some significant differences with the `HP_ALIGN` pragma, including uniform bitfield alignment, uniform struct and union alignment, and the lack of `PUSH` and `POP` functionality.

For complete details on the use of `HP_ALIGN` and `PACK` pragmas, refer to Chapter 2, “Storage and Alignment Comparisons,” in the *HP C/HP-UX Programmer's Guide*.

Listing Pragmas

The listing pragmas introduced in this section are discussed in more detail in “The Listing Facility” on page 279.

LINES Pragma

```
#pragma LINES linenum
```

Sets the number of lines per page to *linenum*. The default is 63. The minimum number of lines per page is 20.

WIDTH Pragma

```
#pragma WIDTH pagewidth
```

Sets the width of the page to *pagewidth*. The default is 80 columns. The minimum number of columns per page is 50. Place the WIDTH pragma before any TITLE or SUBTITLE pragmas. The width of the title and subtitle fields varies with the page width.

TITLE Pragma

```
#pragma TITLE "string"
```

Makes *string* the title of the listing. *String* can have a length of up to 44 characters less than the page width (additional characters are truncated with no warning). The default is no title.

SUBTITLE Pragma

```
#pragma SUBTITLE "string"
```

Makes *string* the subtitle of the listing. *String* can have a length of up to 44 characters less than the page width (additional characters are truncated with no warning). The default is no subtitle.

PAGE Pragma

```
#pragma PAGE
```

Causes a page break and begins a new page.

LIST Pragma

```
#pragma LIST { ON }
```

```
#pragma LIST { OFF }
```


Turns listing functionality ON or OFF when used with the `-Wc`, `-L` command line option. The default is ON. Use this pragma to exclude source lines you do not need to list such as include files.

AUTOPAGE Pragma

```
#pragma AUTOPAGE { ON }  
#pragma AUTOPAGE { OFF }
```

ON causes a page break after each function definition. If the pragma is declared without specifying either the ON or OFF option, then page breaks are generated. If the pragma is not used then no page breaks are generated.

Optimization Pragmas

For additional information on the following optimization pragmas see Chapter 4, “Optimizing C Programs,” and Chapter 8, “Threads and Parallel Processing,” of the *HP C/HP-UX Programmer's Guide*.

ALLOCS_NEW_MEMORY Pragma

```
#pragma ALLOCS_NEW_MEMORY functionname1, . . . , functionnamen
```

States that the function *functionname* returns a pointer to new memory that it allocates or a routine that it calls allocates. This pragma provides additional information to the optimizer which results in more efficient code. (See the *HP C/HP-UX Programmer's Guide* for additional information.)

FLOAT_TRAPS_ON Pragma

```
#pragma FLOAT_TRAPS_ON { functionname,...functionname }  
#pragma FLOAT_TRAPS_ON { _ALL }
```

Informs the compiler that you may have enabled floating-point trap handling. When the compiler is so informed, it will not perform loop invariant code motion (LICM) on floating-point operations in the functions named in the pragma. This pragma is required for proper code generation when floating-point traps are enabled and the code is optimized.

The `_ALL` parameter specifies that loop invariant code motion should be disabled for all functions within the compilation unit.

[NO]INLINE Pragma

```
#pragma INLINE [ functionname1 , . . . , functionnamen ]
```

```
#pragma NOINLINE [ functionname1 , . . . , functionnamen ]
```

Enables (or disables) inlining of functions. If particular functions are specified with the pragma, they are enabled (or disabled) for inlining. If no functions are specified with the pragmas, all functions are enabled (or disabled) for inlining. Refer to the HP *C/HP-UX Programmer's Guide* for details and examples.

[NO]PTRS_STRONGLY_TYPED Pragma

```
#pragma PTRS_STRONGLY_TYPED BEGIN
```

```
#pragma PTRS_STRONGLY_TYPED END
```

```
#pragma NOPTRS_STRONGLY_TYPED BEGIN
```

```
#pragma NOPTRS_STRONGLY_TYPED END
```

Specifies when a subset of types are type-safe, providing a finer level of control that `+O[no]ptrs_strongly_typed`. The pragma will take precedence over the command line option, although sometimes both are required. Refer to the HP *C/HP-UX Programmer's Guide* for details and examples.

NO_SIDE_EFFECTS Pragma

```
#pragma NO_SIDE_EFFECTS functionname1 , . . . , functionnamen
```

States that *functionname* and all the functions that *functionname* calls will not modify any of a program's local or global variables. This pragma provides additional information to the optimizer which results in more efficient code. See the HP *C/HP-UX Programmer's Guide* for further information.

Shared Library Pragma

This section describes a pragma for shared libraries. For detailed information on shared libraries, see the *HP-UX Linker and Libraries Online User Guide*.

HP_SHLIB_VERSION Pragma

```
#pragma HP_SHLIB_VERSION "mm/[yy]yy"
```

Assigns a version number to a shared library module. This enables you to store multiple versions of a subroutine in a shared library.

The version number is specified by *mm/[yy]yy*. *mm* represents the month, and must be from 1 to 12. *[yy]yy* represents the year, either in 2 digits or 4 digits. If the 2 digit form is used, it must be from 90 to 99, and will be interpreted as 1990 to 1999. The 4 digit form must be from 1990 to 7450.

This pragma provides a way to guard against unexpected side effects when a shared library is updated. You can put multiple versions of a routine in the library and ensure that programs use the correct version. The date in the `SHLIB_VERSION` pragma provides the version number. Programs call the version in the shared library with a date less than or equal to the date the program was linked.

The version number should only be incremented when changes made to a subroutine make the new version of the subroutine incompatible with previous versions.

FastCall Pragmas

The compiler directives described in this section are designed to speed-up shared library calls. For more information about how these pragmas help shared library performance, see “Improving Shared Library Performance” on page 138.

HP_DEFINED_EXTERNAL Pragma

```
#pragma HP_DEFINED_EXTERNAL sym1, sym2, ...
```

The externally defined symbol pragma specifies that the designated symbols are imported from another load module (program file or shared library). Note that this pragma currently works in 32-bit mode only. For 64-bit mode, see the option `+Oextern`.

HP_LONG_RETURN Pragma

```
#pragma HP_LONG_RETURN func1, func2, ...
```

The long return sequence pragma specifies that the named procedures can return directly across a space, without needing to return through an export stub. The main goal of this pragma is to eliminate export stubs, and better enable inlining of import stubs and `$$dyncall` functionality for indirect calls.

HP_NO_RELOCATION Pragma

```
#pragma HP_NO_RELOCATION func1, func2, ...
```

The no parameter/return relocation pragma is used to suppress unnecessary floating point argument relocation. When used, the burden is on the caller and callee to have their argument and their return values agree in which register files they are passed.

Running HP C Programs

After a program is successfully linked, it is in executable form. To run the program, enter the executable filename (either `a.out` or the name following the `-o` option).

Compiling and Running HP C Programs
Running HP C Programs

10

**HP C/HP-UX Implementation
Topics**

This chapter describes topics that are specific to programming in C on HP 9000 computers.

Data Types

Data types are implemented in HP C/HP-UX as follows:

- The `char` type is signed.
- All types can have the `register` storage class, although it is only honored for scalar types. Ten register declarations per function are honored.
- The signed integer types are represented internally using twos complement form.
- Structures and unions start and end on an alignment boundary which is that of their most restrictive member.
- The `long long` data type cannot be used to declare an array's size.
- The `long long` data type is available only under `-Ac`, `-Aa +e`, and `-Ae` compilation modes.

Table 10-1 lists the sizes and ranges of different HP C/HP-UX data types.

Refer to the *HP C/HP-UX Programmer's Guide* for comparisons of data storage and alignment on the following computer systems:

- HP 3000 Series 900
- HP 3000/V
- HP 9000 Series 700/800

Table 10-1 HP C/HP-UX Data Types

Type	Bits	Bytes	Low Bound	High Bound	Comments
char	8	1	-128	127	Character
signed char	8	1	-128	127	Signed integer
unsigned char	8	1	0	255	Unsigned integer
short	16	2	-32,768	32,767	Signed integer
unsigned short	16	2	0	65,535	Unsigned integer
int	32	4	-2,147,483,648	2,147,483,647	Signed integer
unsigned int	32	4	0	4,294,967,295	Unsigned integer
long (ILP32)	32	4	-2,147,483,648	2,147,483,647	Signed integer
long (LP64)	64	8	-2^{63}	$2^{63} - 1$	Signed integer
long long	64	8	-2^{63}	$2^{63} - 1$	Signed integer
unsigned long (ILP32)	32	4	0	4,294,967,295	Unsigned integer
unsigned long (LP64)	64	8	0	$2^{64} - 1$	Unsigned integer
unsigned long long	64	8	0	$2^{64} - 1$	Unsigned integer
float	32	4	See (a) below.	See (b) below.	Floating-point
double	64	8	See (c) below.	See (d) below.	Floating-point
long double	128	16	See (e) below.	See (f) below.	Floating-point
enum	32	4	-2,147,483,648	2,147,483,647	Signed integer

Comments

In the following comments, the low bounds of `float`, `double`, and `long double` data types are given in their *normalized* and *denormalized* forms. Normalized and denormalized refer to the way data is stored. Normalized numbers are represented with a greater degree of accuracy than denormalized numbers. Denormalized numbers are very small numbers represented with fewer significant bits than normalized numbers.

- a. Least normalized: 1.17549435E-38F
Least denormalized: 1.4012985E-45F
- b. 3.40282347E+38F
- c. Least normalized: 2.2250738585072014E-308
Least denormalized: 4.9406564584124654E-324
- d. 1.7976931348623157E+308
- e. Least normalized:
3.3621031431120935062626778173217526026E-4932L

Least denormalized:
6.4751751194380251109244389582276465525E-4966L
- f. 1.1897314953572317650857593266280070162E+4932L

Bit-Fields

- Bit-fields in structures are packed from left to right (high-order to low-order).
- The high order bit position of a “plain” integer bit-field is treated as a sign bit.
- Bit-fields of types `char`, `short`, `long`, `long long`, and `enum` are allowed.
- The maximum size of a bit-field is 64 bits.
- If a bit-field is too large to fit in the current word, it is moved to the next word.
- The range of values in an integer bit-field are:
 - 2,147,483,648 to 2,147,483,647 for 32-bit signed quantities
 - 0 to 4,294,967,295 for 32-bit unsigned quantities
 - 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 for 64-bit signed quantities
 - 0 to 18,446,744,073,709,551,615 for 64-bit unsigned quantities
- Bit-fields in unions are allowed only in ANSI mode.

IEEE Floating-Point Format

The internal representation of floating-point numbers conforms to the IEEE floating-point standard, ANSI/IEEE 754-1985, as shown in figure 10-1.

Figure 10-1 Internal Representation of Floating-Point Numbers



The *s* field contains the sign of the number. The *exp* field contains the biased exponent ($exp = E + bias$, where *E* is the real exponent) of the number. The values of *bias* and the maximum and minimum values of the unbiased exponent appear in the following table:

Table 10-2 Floating-point Bias, and Unbiased Exponents Values

	float	double	long double
bias	+127	+1023	+16383
E_{max}	+127	+1023	+16383
E_{min}	-126	-1022	-16382

$E_{min}-1$ is used to encode 0 and denormalized numbers.

$E_{max}+1$ is used to encode infinities and NaNs.

NaNs are binary floating-point numbers that have all ones in the exponent and a nonzero fraction. NaN is the term used for a binary floating-point number that has no value (that is, “Not A Number”).

If E is within the range

$$E_{\min} \leq E \leq E_{\max}$$

the mantissa field contains the number in a normalized form, preceded by an implicit 1 and binary point.

In accordance with the IEEE standard, floating-point operations are performed with traps not enabled, and the result of such an operation is that defined by the standard. This means, for example, that dividing a positive finite number by zero will yield positive infinity, and no trap will occur. Dividing zero by zero or infinity by infinity will yield a NaN, again with no trap. For a discussion of infinity arithmetic and operations with NaNs, in the context of the IEEE standard, see the *HP Precision Architecture and Instruction Set Reference Manual* (HP part number 09740-90014).

For detailed information about floating-point arithmetic on HP-UX, how HP-UX implements the IEEE standard, and the HP-UX math libraries, see the *HP-UX Floating Point Guide*.

Note that infinities and NaNs propagate through a sequence of operations. For example, adding any finite number to infinity will yield infinity. An operation on a NaN will yield a NaN. This means that you may be able to perform a sequence of calculations and then check just the final result for infinity or NaN.

The HP-UX math library provides routines for determining the class of a floating point number. For example, you can determine if a number is infinity or NaN. See the *HP-UX Reference* for descriptions of the functions `fpclassify`, `fpclassifyf`, `isinf`, and `isnan`.

Lexical Elements

- Identifiers: 255 characters are significant in internal and external names.
- Character Constants: Any character constant of more than one character produces a warning. The value of an integral character constant containing more than one character is computed by concatenating the 8-bit ASCII code values of the characters, with the leftmost character being the most significant. For example, the character constant 'AB' has the value $256 * 'A' + 'B' = 256 * 65 + 66 = 16706$. Only the rightmost four characters participate in the computation.
- The case of alphabetic characters is always significant in external names.
- The execution character set and the source character set are both ASCII.
- Nonprinting characters in character constants and string literals must be represented as escape sequences.

Structures and Unions

Structure or union references that are not fully qualified (see example below) are flagged with an error by the compiler.

```
struct{
    int j;
    struct {int i;}in;
} out;
out.i=3;
```

The correct statement for the example above is `out.in.i = 3;`

Type Mismatches in External Names

It is illegal to declare two externally visible identifiers of different types with the same name in separately compiled translation units. The linker might not diagnose such a mismatch.

Expressions

The value of an expression that overflows or underflows is undefined, except when the operands are unsigned.

Pointers

- Pointers to functions should not be compared using relational operators because the pointers represent external function labels and not actual addresses.
- Dereferencing a pointer that contains an invalid value results in a trap if the address references protected memory or if the address is not properly aligned for the object being referenced.
- A declaration of a pointer to an undefined structure tag is allowed, and the tag need not be defined in the source module unless the pointer is used in an expression.

Maximum Number of Dimensions of an Array

Arrays can have up to 252 dimensions.

Scope of extern Declarations

Identifiers for objects and functions declared within a block and with the storage class `extern` have the same linkage as any visible declaration with file scope. If there is no visible declaration with file scope, the identifier has external linkage, and the definition remains visible until the end of the translation unit.

However, because this is an extension to ANSI C, a warning will be issued on subsequent uses of the identifier if the absence of this extended visibility could cause a change in behavior on a port to another conforming implementation.

Conversions Between Floats, Doubles, and Long Doubles

- When a long double is converted to a double or float, or when a double is converted to a float, the original value is rounded to the nearest representable value as the new type. If the original value is equally close to two distinct representable values, then the value chosen is the one with the least significant bit equal to zero.
 - Conversions between floating-point types involve a change in the exponent, as well as the mantissa. It is possible for such a conversion to overflow.
-

Statements

- The types of `switch` expressions and their associated `case` label constants do not need to match. Integral types can be mixed.
- All expressions of integral types are allowed in `switch` statements.

Preprocessor

- The maximum nesting depth of `#include` files is 35.
- For include files that are enclosed in double quotes and do not begin with a `/`, the preprocessor will first search the current directory, then the directory named in the `-I` option, and finally, in the standard include directory `/usr/include`.
- For include files that are enclosed in `<` and `>` signs, the search path begins with the directory named in the `-I` option and is completed in the standard include directory, `/usr/include`. The current directory is not searched.

Library Functions and Header Files

This section describes the implementation of library functions in HP C/HP-UX. For complete information about library functions on HP C/HP-UX, see the *HP-UX Reference* manual and *HP-UX Linker and Libraries Online User Guide*.

The Math Library

When using any of the mathematical functions in the `<math.h>` header, you must include the `-lm` flag on the `cc` or `ld` command when linking. This will cause the linker to link in the appropriate math library.

NOTE

C math libraries have traditionally used a function called `matherr`, which was required by the SVID2 specification but is not specified by ANSI C, SVID3, or XPG4.2. In the HP-UX math library, the SVID2 `matherr` function exists under the names `matherr` and `_matherr`. These functions are still provided in `libm.a` to assist in supporting old programs. Executables built at HP-UX releases 10.0 through 10.20 that use `matherr` or `_matherr` will continue to run at the next release of HP-UX. However, these functions are obsolete and will not be supported for newly compiled or linked programs at the next release of HP-UX.

Refer to the *HP-UX Floating-Point Guide* for further details.

Other Library Functions

- **longjmp:** Because HP C/HP-UX can place automatic variables in registers, you cannot rely on their values if they are changed between the `setjmp` and `longjmp` functions.
- **setjmp:** There are no restrictions on when calls to `setjmp` can be made.

The varargs Macros

The `varargs` macros allow accessing arguments of functions where the number and types of the arguments can vary from call to call.

NOTE

The `<varargs.h>` header has been superseded by the standard header `<stdarg.h>`, which provides all the functionality of the `varargs` macros. The `<varargs.h>` header is retained for compatibility with pre-ANSI compilers and earlier releases of HP C/HP-UX.

To use `varargs`, a program must include the header `<varargs.h>`. A function that expects a variable number of arguments must declare the first variable argument as `va_alist` in the function declaration. The macro `va_dcl` must be used in the parameter declaration.

A local variable should be declared of type `va_list`. This variable is used to point to the next argument in the variable argument list.

The `va_start` macro is used to initialize the argument pointer to the initial variable argument.

Each variable argument is accessed by calling the `va_arg` macro. This macro returns the value of the next argument, assuming it is of the specified type, and updates the argument pointer to point to the next argument.

The `va_end` macro is provided for consistency with other implementations; it performs no function on the HP 9000 Series 800 computers. The following example demonstrates the use of the `<varargs.h>` header:

Example

```
#include <varargs.h>
#include <stdio.h>

enum arglisttype {NO_VAR_LIST, VAR_LIST_PRESENT};
enum argtype     {END_OF_LIST, CHAR, DOUB, INT, PINT};

int foo (va_alist)
va_dcl /* Note: no semicolon */
{
    va_list ap;
    int a1;
    enum arglisttype a2;
```

HP C/HP-UX Implementation Topics

The varargs Macros

```
enum argtype ptype;
int i, *p;
char c;
double d;

/* Initialize the varargs mechanism */
va_start(ap);

/* Get the first argument, and arg list flag */
a1 = va_arg (ap, int);
a2 = va_arg (ap, enum arglisttype);

printf ("arg count = %d\n", a1);

if (a2 == VAR_LIST_PRESENT) {
/* pick up all the arguments */
do {
/* get the type of the argument */
ptype = va_arg (ap, enum argtype);

/* retrieve the argument based on the type */
switch (ptype) {
case CHAR: c = va_arg (ap, char);
printf ("char = %c\n", c);
break;

case DOUB: d = va_arg (ap, double);
printf ("double = %f\n", d);
break;

case PINT: p = va_arg (ap, int *);
printf ("pointer = %x\n", p);
break;

case INT : i = va_arg (ap, int);
printf ("int = %d\n", i);
break;

case END_OF_LIST :
break;

default: printf ("bad argument type %d\n", ptype);
ptype = END_OF_LIST; /* to break loop */
break;
} /* switch */
} while (ptype != END_OF_LIST);
}

/* Clean up */
va_end (ap);
}

main()
{
int x = 99;

foo (1, NO_VAR_LIST);
foo (2, VAR_LIST_PRESENT, DOUB, 3.0, PINT, &x, END_OF_LIST);
}
```

C9X standard macros

The C9X standard-compliant version of "variadic" or variable argument (varargs) macro notation has been added to the HP ANSI C compiler. The notation for C9X standard and for the GNU version of varargs is very similar.

If you have coded your macros to the GNU standards, you can expect GNU-style behavior using the HP ANSI C compiler. If you have coded your macros to C9X standards, you can expect C9X-style behavior.

Usage differences

In the HP ANSI C compiler, an ellipsis (...) at the end of the parameter list in a macro definition indicates that the macro takes a variable number of arguments, or is variadic in nature. The ellipsis should be the final parameter in C9X style and should immediately follow the final parameter in GNU style.

The last parameter in a variable argument definition is referred to as the variable argument parameter. In GNU terminology, this is known as the "rest" parameter. In a C9X style definition, the variable argument parameter does not explicitly appear in the parameter list but is referred to by the identifier `__VA_ARGS__` in its replacement text.

In the use of a variable arguments macro, a one-to-one correspondence is set up between the arguments in use and those in its definition. This is up to, but not including, the variable argument parameter. The rest of the arguments in the use of the macro definition are referred to as the trailing arguments. For purposes of expanding the macro, the entire combination of the trailing arguments (including the separating commas) is substituted for the variable argument parameter in the replacement text.

There are minor usage differences between how C9X and GNU specify variable argument macros are defined:

Table 10-3

How C9X and GNU define a variable argument macro

C9X

```
#define foo(f, ...) printf (f, __VA_ARGS__)
```

GNU

```
#define foo(f, s...) printf(f, s)
```

- In the GNU style, the name of the variable parameter *s* precedes the ellipsis in the parameter list.
- In the C9X standard, the identifier `__VA_ARGS__` refers to the variable arguments.
- The identifier `__VA_ARGS__` can only occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the arguments.

Variable names

Variable names are also handled slightly different by C9X and GNU.

Table 10-4

How C9X and GNU refer to varargs in macro replacement text

C9X	GNU
Specified by the identifier <code>__VA_ARGS__</code> .	Name appears in the replacement text.

For example:

- `__VA_ARGS__` in the replacement text indicates the variable name in the following C9X-style code:

```
printf(f, __VA_ARGS__)
```

- *s* in the replacement text indicates the variable name in the following GNU-style code:

```
printf(f, s)
```

How HP C implements GNU and C9X macros

If you intend to use GNU style variable argument macros in HP C, note that you can make the concatenation operator `##` prevent syntax errors from occurring when the variable argument comes in as empty (the null string). However, you can also insert whitespace to the left of the left operand of `##` to more accurately specify the intended left operand.

For example, if you use

```
#define foo(f, s...) printf(f, s)
```

Then the macro "call"

```
foo("Hello world.\n");
```

results in the expansion

```
printf("Hello world.\n",);
```

(note the comma ",") causing a syntax error.

GNU provides the following workaround for this kind of a situation. If you use:

```
#define foo(f, s...) printf(f, ## s)
```

If the variable parameter 's' is non-null, if for example, you use:

```
foo("%s %d\n", "Cycles", "1024");
```

the result is

```
printf("%s %d\n", "Cycles", "1024");
```

as the expansion as you would expect.

However, if 's' is null, this erases the comma to the left of the '##' in the macro definition and resulting expansion is:

```
printf("Hello world.\n");
```

Note that the comma is gone.

In order to get the same behavior in HP C, you must insert a space to the left of the comma to make it clear to the preprocessor that the comma is the left operand of the '##' operator. Thus your definition for the macro 'foo' is:

```
#define foo(f, s...) printf(f , ## s)
```

(Note the space to the left of the '##' operator in the macro definition.)

If the space is not inserted, the left operand of the '##' operator is understood to be:

```
printf(f,
```

Because there is no parameter by that name for 'foo', it is erased.

HP Specific Type Qualifiers

See “HP-Specific Type Qualifiers” on page 42.

Location of Files

Table 10-5 lists the location of the HP C files.

Table 10-5 **Location of Files**

File or Library	Location
Driver	/opt/ansic/bin/cc /opt/ansic/bin/c89
Preprocessor	/opt/langtools/lbin/cpp (Compatibility mode) /opt/langtools/lbin/cpp.ansi (ANSI mode)
Compiler	/opt/ansic/lbin/ccom
Assembler	/usr/ccs/bin/as
Linker	/usr/ccs/bin/ld - a “wrapper” that determines if the link is a 32-bit link or a 64-bit link. It then invokes one of /usr/ccs/lbin/ld32 or /usr/ccs/lbin/ld64.
64-bit dynamic loader	/usr/lib/pa20_64/dld.sl
Advanced Optimizing Code Generator	/opt/langtools/lbin/ucomp (32-bit mode) /opt/langtools/lib/libcomp.sl (64-bit mode)
C libraries (libc)	/usr/lib/libc.a ¹ /lib/libc.sl
Debugging Aid Routines	/opt/langtools/lib/end.o
64-bit libraries	/opt/langtools/lib/pa20_64/libl.a /opt/langtools/lib/pa20_64/liby.a

File or Library	Location
libraries for threads	/opt/langtools/lib/libl_thread.a /opt/langtools/lib/pa20_64/libl_thread.a
manpages	/opt/ansic/share/man/man1.Z/cb.1 (English)
	/opt/ansic/share/man/man1.Z/cc.1 (English)
	/opt/ansic/share/man/man1.Z/cflow.1 (English)
	/opt/ansic/share/man/man1.Z/cxref.1 (English)
	/opt/ansic/share/man/man1.Z/lint.1 (English)
	/opt/ansic/share/man/man1.Z/protogen.1 (English)
	/opt/langtools/share/man/man1.Z/cpp.1 (English)
	/opt/langtools/share/man/man1.Z/lex.1 (English)
	/opt/langtools/share/man/man1.Z/yacc.1 (English)
Message Catalogs	/opt/ansic/lib/nls/msg/C/cc.cat (English)

HP C/HP-UX Implementation Topics
 Location of Files

File or Library	Location
	/opt/ansic/lib/nls/msg/C/cb.cat (English)
	/opt/ansic/lib/nls/msg/C/cflow.cat (English)
	/opt/ansic/lib/nls/msg/C/cxref.cat (English)
	/opt/ansic/lib/nls/msg/C/lint.cat (English)
	/opt/ansic/lib/nls/msg/C/protogen.cat (English)
	/opt/langtools/lib/nls/msg/C/cpp.cat (English)
	/opt/langtools/lib/nls/msg/C/lex.cat (English)
	/opt/langtools/lib/nls/msg/C/yacc.cat (English)
	/opt/langtools/lib/nls/msg/C/libmp.cat (English)
Normal startup	opt/langtools/lib/crt0.o
PBO startup	/opt/langtools/lib/icrt0.o
PBO Shared Libraries startup ²	/opt/langtools/lib/scrt0.o
gprof startup	/opt/langtools/lib/gcrt0.o
prof startup	/opt/langtools/lib/mcrt0.o
crt0.o for +Oparallel	/opt/langtools/lib/mpcrt0.o
crt0.o for +Oparallel and gprof	/opt/langtools/lib/mpgcrt0.o
Assembler prefix file	/usr/lib/pcc_prefix.s
Profiled C library	/usr/lib/libp/libc.a
Temporary files	/var/tmp ³

File or Library	Location
Math libraries ^{4,5}	/usr/lib/libm.a (PA-RISC 1.0, archive)
	/usr/lib/libp/libm.a (a profiled version of /usr/lib/libm.a)
	/usr/lib/libm.sl (PA-RISC 1.0, shared)
	/usr/lib/pa1.1/libm.a (PA-RISC 1.1, archive) ⁶
Parallel Runtime Library	/opt/langtools/lib/libmp.a
Standard Include Files	/usr/include
Online Help	/opt/ansic/html/guide/C
C Tools	/opt/ansic/bin
	/opt/ansic/lbin
	/opt/langtools/bin
	/opt/langtools/lbin

¹You can change the search path for `libc.a` through the `-Ldir` option. See Chapter 9, “Compiling and Running HP C Programs,” on page 197 for details.

²See the *HP-UX Linker and Libraries Online User Guide* for more information on PBO of shared libraries.

³You can change the default location for the temporary files used and created by the C compiler by setting the environment variable `TMPDIR`. If the compiler cannot write to `$TMPDIR`, it uses the default location `/var/tmp`. See the *HP-UX Reference* for details.

⁴The appropriate library file is automatically selected based upon the options selected on the command line.

⁵The `libM` library, which formerly supported XPG and POSIX while the `libm` library supported SVID, is obsolete now that these standards are compatible. The various versions of `libM` now exist only as symbolic links to the corresponding versions of `libm`. The symbolic links will eventually disappear.

⁶For performance reasons, HP provides the PA1.1 version of `libm.a` only as an archive library. For information about performance issues related to shared and archive libraries, refer to the *HP-UX Floating Point Guide*.

An intrinsic is an external routine that can be called by HP C or any language that the HP-UX operating system supports. The intrinsic can be written in any supported language. However, its formal parameters must be of data types that have counterparts in HP C. This chapter describes the use of intrinsic functions in HP C programs.

Intrinsics are used much like library functions, except that users may write their own intrinsic routines, and then create an `INTRINSIC_FILE`. This file contains information about the number and type of parameters of the intrinsic. It is similar to a C header file. Additionally, the information in the `INTRINSIC_FILE` is accessible by other compilers, such as Pascal or Fortran. This will allow a single interface for multiple languages.

The body of an intrinsic routine is put into a library that contains the bodies of other intrinsics. Then, the declarations are put into an `INTRINSIC_FILE`. When calling an intrinsic from a program, `#pragma INTRINSIC` is used to specify the name of the intrinsic routine, and `#pragma INTRINSIC_FILE` is used to specify the location of the declaration of the intrinsic routine. Since the body of the intrinsic routine actually resides in a user-built library, that library must be linked in explicitly.

INTRINSIC Pragma

The `INTRINSIC` pragma allows you to declare an external function as an intrinsic. This pragma has the following format:

```
#pragma INTRINSIC intrinsic_name1 [user_name] [intrinsic_name2  
user_name]]...
```

where:

intrinsic_name is the name of the intrinsic you want to declare.

user_name is a valid C identifier. If specified, you can use this name to invoke the intrinsic from the source program.

Examples

```
#pragma INTRINSIC FOPEN  
#pragma INTRINSIC FCLOSE myfclose  
#pragma INTRINSIC FCHECK, FGETINFO  
#pragma INTRINSIC FWRITE mpe_fwrite, FREAD mpe_fread
```

The first example shows how to declare the `FOPEN` intrinsic. The second example shows how to declare `FCLOSE`; you must call it by the name `myfclose` in your program. The third and fourth examples each declare two intrinsics. The fourth provides alternative names for the intrinsics.

When you designate an external function as an intrinsic, the compiler refers to a special file (called an intrinsic file) to determine the function type, the number of parameters, and the type of each parameter. The compiler then uses this information to perform the necessary conversions and insertions to invoke the routine, or to issue warnings and errors if proper invocation is not possible.

Specifically, for intrinsic calls, the HP C compiler does the following:

- Converts all value parameters to the type expected by the intrinsic function. Conversions are performed as if an assignment is done from the value to the formal parameter. This is known as *assignment conversion*. If a value cannot be converted, an error message is issued.
- Converts addresses passed as reference parameters to the proper address type. This essentially means that short addresses are converted to long addresses as required by the intrinsic function. An integer value of zero is considered a legal value (NULL) for any address parameter.

- Allows missing arguments in the call to the intrinsic if the intrinsic defines default values for those parameters. The compiler supplies the default values for the missing arguments, or issues a diagnostic if there is no defined default value. Missing arguments are allowed within an argument list or at the end of an argument list.
- Issues an error message if there are too many arguments.
- Inserts “hidden” arguments required by Pascal routines that have ANYVAR parameters (size is hidden), or that are EXTENSIBLE (parameter count is hidden). See the *HP Pascal Programmer's Guide* for more information.

Remember that C does not normally do any parameter counting, converting, or checking. So, if you attempt to declare an intrinsic using a standard C declaration rather than using the `#pragma INTRINSIC` statement, none of the above checks, conversions, or insertions are done. The address of an intrinsic can be taken, but if a call is made using a pointer to the intrinsic, the above checks are not performed. The intrinsic call then degenerates into a normal C function call.

To ensure that all calls are handled as expected, the intrinsic pragma should declare the name of the intrinsic using an identifier with the identical case that is used by the function calls in the program, especially if the optional user name is not specified. No other functions in the program should have the same name as any intrinsic that is declared, regardless of the case. This is because the actual run time symbol used to call an intrinsic is not necessarily the same case as the identifier used to declare that intrinsic.

INTRINSIC_FILE Pragma

The `INTRINSIC_FILE` pragma specifies the path of a file in which the compiler can locate information about intrinsic functions. This pragma has the following format:

```
#pragma INTRINSIC_FILE "path"
```

where *path* is the fully qualified path of a file. The compiler will look in this file for information about intrinsics declared using the `INTRINSIC` pragma. If you do not specify a full path name, the compiler searches in your current directory.

If you do not use the `INTRINSIC_FILE` pragma, the compiler looks in a file called `/usr/lib/sysintr`. You need to use the `INTRINSIC_FILE` pragma only if you are building your own intrinsic files using the HP Pascal compiler and you must specify a file other than the default. Refer to the *HP Pascal Programmer's Guide* for information about building your own intrinsic files.

The compiler searches in the specified file until another `INTRINSIC_FILE` pragma is encountered. To return the search to `/usr/lib/sysintr`, specify the `INTRINSIC_FILE` pragma with a null string, as shown below:

```
#pragma INTRINSIC_FILE ""
```

Here are some examples of `INTRINSIC_FILE` and `INTRINSIC` pragmas:

```
#pragma INTRINSIC FOPEN, FCLOSE, FREAD /* /usr/lib/sysintr used*/  
#pragma INTRINSIC_FILE "myintr"  
#pragma INTRINSIC mytest1, mytest2 /* myintr used */  
#pragma INTRINSIC_FILE ""  
#pragma INTRINSIC FCHECK, FGETINFO /* /usr/lib/sysintr used */
```

In the first example above, the compiler searches the default file for information about the `FOPEN`, `FCLOSE`, and `FREAD` intrinsics. The second pragma specifies a different file for the compiler to search, `myintr`. The compiler looks for this file in the current directory. The third pragma declares two intrinsics, `mytest1` and `mytest2`, which must be described in `myintr`. The fourth pragma returns the search to `/usr/lib/sysintr`, where `FCHECK` and `FGETINFO` are sought when the fifth pragma is encountered.

12

The Listing Facility

The HP C compiler generates a listing whenever a program is compiled with the `+L` option. This listing is sent to standard output (`stdout`), and it can be redirected to a file using the shell redirection facility.

Listing Format

The listing consists of the following information:

- A banner on the top of each page.
- A line number for each source line.
- The nesting level for each statement or declaration.

There are two styles of listing available: *compatibility mode* and *ANSI mode*.

Compatibility Mode

In compatibility mode, the text of the listing is the output of the preprocessor, after macro substitution with `#include` files inserted.

ANSI Mode

In ANSI mode, the text of the listing is the original version of the source file, before macro substitution; `#include` files are inserted. To produce the non-ANSI style listing, compile with the `+LP` option instead of the `+L` option.

NOTE

The `+LP` option only has this effect when it is used in conjunction with the `-Aa` option.

In either mode, comments are stripped from the listing (unless the `-C` option is specified).

Listing Pragmas

The listing facility provides a number of pragmas to control various features of the listing format. The available pragmas are described below.

```
#pragma LINES linenum
```

Sets the number of lines per page to *linenum*. Default is 63. Minimum number is 20 lines.

```
#pragma WIDTH pagewidth
```

Sets the width of the page to *pagewidth*. Default is 80 columns. Minimum number is 50 columns.

NOTE

If the WIDTH pragma is being used, put it before any TITLE or SUBTITLE pragmas, since the title and subtitle field widths vary with the page width.

```
#pragma TITLE "string"
```

Sets the page title to *string*. *string* is truncated without warning to 44 characters less than the page width. Default is the empty string.

```
#pragma SUBTITLE "string"
```

Sets the page subtitle to *string*. *string* is truncated without warning to 44 characters less than the page width. Default is the empty string.

NOTE

The TITLE and SUBTITLE pragmas do not take effect until the second page, because the banner on the first page appears before the pragmas.

```
#pragma PAGE
```

Causes a page break and the start of a new page.

```
#pragma AUTOPAGE { ON }
```

```
#pragma AUTOPAGE { OFF }
```

Causes a page break after each function definition. Default is OFF.

```
#pragma LIST { ON }
```

```
#pragma LIST { OFF }
```

Turns the listing ON/OFF. The default is ON. Use this pragma as a toggle to turn listing off around any source lines that you do not want to be listed, such as include files.

Listing Options

Two compiler options are provided to write additional information to the listing. The `+m` option is used to generate identifier maps. The `+o` option is used to generate code offsets.

Identifier Maps

When the `+m` option is specified, the compiler produces a series of identifier maps, grouped by function. The map shows the declared identifiers, their storage class, type, and address or constant value.

The first column of the map lists, in alphabetical order, the initial 20 characters of all the identifiers declared in the function. Member names of structures and unions appear indented under the structure or union name.

The second column displays the storage class of each identifier. The compiler distinguishes the following storage classes:

<code>auto</code>	<code>external definition</code>	<code>static</code>
<code>constant</code>	<code>member</code>	<code>typedef</code>
<code>external</code>	<code>register</code>	

The third column shows the type of the identifier. The types include:

<code>array</code>	<code>int</code>	<code>unsigned char</code>
<code>char</code>	<code>long int</code>	<code>unsigned int</code>
<code>double</code>	<code>long long int</code>	<code>unsigned long</code>
<code>enum</code>	<code>short int</code>	<code>unsigned long long</code>
<code>float</code>	<code>struct</code>	<code>unsigned short</code>
<code>function</code>	<code>union</code>	<code>void</code>

The fourth column indicates the relative register location of an identifier. Members of a union type are in the form `W @ B`, where `W` is the byte offset and `B` is the bit offset within the word. Both offsets are given in hexadecimal notation.

Example

```
main()
{
    enum colors {red, green, blue} this_color;
    struct SS {
        char          *name;
        char          sex;
        int           birthdate;
        int           ssn;
        float         gpa;
        struct SS     *previous;
    } pupil_rec;

    union UU {
        int          flag;
        float        average;
    } datum;

    struct SS second_pupil;

    this_color = red;
    pupil_rec.sex = 'm';
    datum.flag = 1;
    second_pupil.gpa = 3.72;
}
```

```
GLOBAL IDENTIFIER MAP
Identifier      Class      Type      Address
main            ext def  int ()    main

LOCAL IDENTIFIER MAPS

main

Identifier      Class      Type      Address
blue            const     enum colors  2
datum          auto      union UU    SP-64
  flag          member    int         0x0 @ 0x0
  average       member    float       0x0 @ 0x0
green          const     enum colors  1
pupil_rec      auto      struct SS   SP-60
  name          member    char *      0x0 @ 0x0
  sex           member    char        0x4 @ 0x0
  birthdate     member    int         0x8 @ 0x0
  ssn           member    int         0xc @ 0x0
  gpa           member    float       0x10 @ 0x0
  previous      member    struct *    0x14 @ 0x0
red            const     enum colors  0
second_pupil   auto      struct SS   SP-88
  name          member    char *      0x0 @ 0x0
  sex           member    char        0x4 @ 0x0
  birthdate     member    int         0x8 @ 0x0
  ssn           member    int         0xc @ 0x0
```

The Listing Facility

Listing Options

```
gpa          member      float          0x10 @ 0x0
previous     member      struct *       0x14 @ 0x0
this_color   auto        enum colors    SP-36
```

Code Offsets

When the `+o` option is specified, the compiler produces a series of the code offsets for each executable statement, grouped by function. Source line numbers are given in decimal notation followed by the associated code address specified in hexadecimal notation. The code address is relative to the beginning of the function.

Example

```
main()
{
    int    j;
    void  func1 ();
    void  func2 ();

    for (j=0; j<50; j++) {
        func1 (j);
        func2 (j);
    }
}

void func1 (i)
{
    int    i;
    while (i &< 50) {
        if (!(i % 5))
            printf ("%d is divisible by 5\n", i);
        i++;
    }
}

void func2 (j)
{
    int    j;

    int    k, m;

    k = j % 10 ? 1 : 0;
    if (k) {
        m = 23;
        k = m * m;
    }
}
```

C O D E O F F S E T S

main "myfile.c"

Line	Offset	Line	Offset	Line	Offset	Line	Offset	Line	Offset
7	8	8	18	9	20				

func1 "myfile.c"

Line	Offset	Line	Offset	Line	Offset	Line	Offset	Line	Offset
17	c	18	14	19	24	20	34		

func2 "myfile.c"

Line	Offset	Line	Offset	Line	Offset	Line	Offset	Line	Offset
30	4	31	20	32	28	33	30		

The Listing Facility
Listing Options

A **Syntax Summary**

This appendix presents a summary of the C language syntax using a variation of the Backus Naur syntax.

Lexical Grammar

Tokens

*token ::= keyword
identifier
constant
string-literal
operator
punctuator*

*preprocessing-token ::=
header-name
identifier
pp-number
character-constant
string-literal
operator
punctuator*

each non-white-space character cannot be one of the above

Keywords

keyword ::= any word from the set:

auto	extern	sizeof
break	float	static
case	for	struct
char	goto	switch
const	if	__thread (HP-UX 10.30 and later)
continue	int	typedef
default	long	union
do	register	unsigned
double	return	void
else	short	volatile
enum	signed	while

Identifiers

identifier ::= *nondigit*
identifier nondigit
identifier digit
identifier dollar-sign

nondigit ::= any character from the set:
_ a b c d e f g h i j k l m n o p
q r s t u v w x y z A B C D E F G
H I J K L M N O P Q R S T U V W X
Y Z

digit ::= any character from the set:
0 1 2 3 4 5 6 7 8 9

dollar-sign ::= the \$ character

Constants

constant ::= *floating-constant*
integer-constant
enumeration-constant
character-constant

floating-constant ::= *fractional-constant* [*exponent-part*] [*floating-suffix*]
digit-sequence *exponent-part* [*floating-suffix*]

fractional-constant ::= [*digit-sequence*] . *digit-sequence*
digit-sequence .

exponent-part ::= e [*sign*] *digit-sequence*
E [*sign*] *digit-sequence*

sign ::= +
-

Syntax Summary

Lexical Grammar

digit-sequence ::=
 digit
 digit-sequence digit

floating-suffix ::=
 f l F L

integer-constant ::=
 decimal-constant [integer-suffix]
 octal-constant [integer-suffix]
 hexadecimal-constant [integer-suffix]

decimal-constant ::=
 nonzero-digit
 decimal-constant digit

octal-constant ::=
 0
 octal-constant octal-digit

hexadecimal-constant ::=
 0x *hexadecimal-digit*
 0X *hexadecimal-digit*
 hexadecimal-constant hexadecimal-digit

nonzero-digit ::= any character from the set:
 1 2 3 4 5 6 7 8 9

octal-digit ::= any character from the set
 0 1 2 3 4 5 6 7

hexadecimal-digit ::= any character from the set
 0 1 2 3 4 5 6 7 8 9
 a b c d e f
 A B C D E F

integer-suffix ::=
 unsigned-suffix [long-suffix]
 length-suffix [unsigned-suffix]

unsigned-suffix ::=
 u U

length-suffix ::=
 long-suffix
 long-long-suffix

long-suffix ::= any character from the set
 l L

long-long-suffix ::= any character from the set
 ll LL Ll lL

enumeration-constant ::= *identifier*

character-constant ::=
 ' *c-char-sequence* '
 L' *c-char-sequence* '

```

c-char-sequence ::=
    c-char
    c-char-sequence c-char

c-char ::=
    any character in the source character set except
    the single quote ('), backslash (\), or new-line character
    escape-sequence

escape-sequence ::=
    simple-escape-sequence
    octal-escape-sequence
    hexadecimal-escape-sequence

simple-escape-sequence ::=
    \'  \"  \?  \\  \ddd  \xdd
    \a  \b  \f  \n  \r  \t  \v

octal-escape-sequence ::=
    \ octal-digit
    \ octal-digit octal-digit
    \ octal-digit octal-digit octal-digit

hexadecimal-escape-sequence ::=
    \x hexadecimal-digit
    hexadecimal-escape-sequence hexadecimal-digit
  
```

String Literals

```

string-literal ::=
    "[s-char-sequence]"
    L"[s-char-sequence]"

s-char-sequence ::=
    s-char
    s-char-sequence s-char

s-char ::=
    any character in the source character set except
    the double-quote ("), backslash (\), or new-line
    character escape-sequence
  
```

Operators

```

operator ::= One selected from:
    [ ] ( ) . ->
    ++ -- & * + - ~ ! sizeof
    / % << >> < > <= >= == != ^ |
    && || ? :
    = *= /= %= += -= <<= >>= &= ^= |=
    , # ##
  
```

Punctuators

punctuator ::= One selected from:
[] () { } * , : = ; ... #

Header Names

header-name ::=
 <*h-char-sequence*>
 "*q-char-sequence*"

h-char-sequence ::=
 h-char
 h-char-sequence h-char

h-char ::=
 any character in the source character set except
 the newline character and >

q-char-sequence ::=
 q-char
 q-char-sequence q-char

q-char ::=
 any character in the source character set except
 the newline character and "

Preprocessing Numbers

pp-number ::=
 digit
 . *digit*
 pp-number digit
 pp-number nondigit
 pp-number e sign
 pp-number E sign
 pp-number .

Phrase Structure Grammar

Expressions

primary-expression ::=
 identifier
 constant
 string-literal
 (*expression*)

postfix-expression ::=
 primary-expression
 postfix-expression [*expression*]
 postfix-expression ([*argument-expression-list*])
 postfix-expression . *identifier*
 postfix-expression -> *identifier*
 postfix-expression ++
 postfix-expression --

argument-expression-list ::=
 assignment-expression
 argument-expression-list , *assignment-expression*

unary-expression ::=
 postfix-expression
 ++ *unary-expression*
 -- *unary-expression*
 unary-operator *cast-expression*
 sizeof *unary-expression*
 sizeof (*type-name*)

unary-operator ::= one selected from
 & * + - ~ !

cast-expression ::=
 unary-expression
 (*type-name*) *cast-expression*

multiplicative-expression ::=
 cast-expression
 multiplicative-expression * *cast-expression*
 multiplicative-expression / *cast-expression*
 multiplicative-expression %% *cast-expression*

additive-expression ::=
 multiplicative-expression
 additive-expression + *multiplicative-expression*
 additive-expression - *multiplicative-expression*

shift-expression ::=
 additive-expression
 shift-expression << *additive-expression*
 shift-expression >> *additive-expression*

Syntax Summary

Phrase Structure Grammar

relational-expression ::=
 shift-expression
 relational-expression < *shift-expression*
 relational-expression > *shift-expression*
 relational-expression <= *shift-expression*
 relational-expression >= *shift-expression*

equality-expression ::=
 relational-expression
 equality-expression == *relational-expression*
 equality-expression != *relational-expression*

AND-expression ::=
 equality-expression
 AND-expression & *equality-expression*

exclusive-OR-expression ::=
 AND-expression
 exclusive-OR-expression ^ *AND-expression*

inclusive-OR-expression ::=
 exclusive-OR-expression
 inclusive-OR-expression | *exclusive-OR-expression*

logical-AND-expression ::=
 inclusive-OR-expression
 logical-AND-expression && *inclusive-OR-expression*

logical-OR-expression ::=
 logical-AND-expression
 logical-OR-expression || *logical-AND-expression*

conditional-expression ::=
 logical-OR-expression
 logical-OR-expression ? *logical-OR-expression* :
 conditional-expression

assignment-expression ::=
 conditional-expression
 unary-expression *assign-operator* *assignment-expression*

assign-operator ::= one selected from the set
 = *= /= %= += -= <<= >>= &= ^= |=

expression ::=
 assignment-expression
 expression , *assignment-expression*

constant-expression ::=
 conditional-expression

Declarations

declaration ::=
 declaration-specifiers [*init-declarator-list*] ;

declaration-specifiers ::=
 storage-class [*declaration-specifiers*]
 type-specifier [*declaration-specifiers*]
 type-qualifier [*declaration-specifiers*]

init-declarator-list ::=
 init-declarator
 init-declarator-list , *init-declarator*

init-declarator ::=
 declarator
 declarator = *initializer*

storage-class-specifier ::=
 typedef
 extern
 static
 auto
 register

type-specifier ::=
 void
 char
 short
 int
 long
 float
 double
 signed
 unsigned
 struct-or-union-specifier
 enum-specifier
 typedef-name

struct-or-union specifier ::=
 struct-or-union [*identifier*] {*struct-declaration-list*}
 struct-or-union *identifier*

struct-or-union ::=
 struct
 union

struct-declaration-list ::=
 struct-declaration
 struct-declaration-list *struct-declaration*

struct-declaration ::=
 specifier-qualifier-list *struct-declarator-list* ;

specifier-qualifier-list ::=
 type-specifier [*specifier-qualifier-list*]
 type-qualifier [*specifier-qualifier-list*]

Syntax Summary

Phrase Structure Grammar

```
struct-declarator-list ::=
    struct-declarator
    struct-declarator-list , struct-declarator

struct-declarator ::=
    declarator
    [declarator] : constant-expression

enum-specifier ::=
    [ type-specifier ] enum [identifier] {enumerator-list}
    [ type-specifier ] enum identifier

enumerator-list ::=
    enumerator
    enumerator-list , enumerator

enumerator ::=
    enumeration-constant
    enumeration-constant = constant-expression

type-qualifier ::=
    const
    noalias
    volatile

declarator ::=
    [pointer] direct-declarator

direct-declarator ::=
    identifier
    ( declarator )
    direct-declarator [ constant-expression ]
    direct-declarator ( parameter-type-list )
    direct-declarator ( [identifier-list] )

pointer ::=
    * [type-qualifier-list]
    * [type-qualifier-list] pointer

type-qualifier-list ::=
    type-qualifier
    type-qualifier-list type-qualifier

parameter-type-list ::=
    parameter-list
    parameter-list , ...

parameter-list ::=
    parameter-declaration
    parameter-list , parameter-declaration

parameter-declaration ::=
    declaration-specifiers declarator
    declaration-specifiers [abstract-declarator]

identifier-list ::=
```

identifier
identifier-list , *identifier*

type-name ::=
 specifier-qualifier-list [*abstract-declarator*]

abstract-declarator ::=
 pointer
 [*pointer*] *direct-abstract-declarator*

direct-abstract-declarator ::=
 (*abstract-declarator*)
 [*direct-abstract-declarator*] [[*constant-expression*]]
 [*direct-abstract-declarator*] ([*parameter-type-list*])

typedef-name ::=
 identifier

initializer ::=
 assignment-expression
 { *initializer-list* }
 { *initializer-list* , }

initializer-list ::=
 initializer
 initializer-list , *initializer*

Statements

statement :=
 labeled-statement
 compound-statement
 expression-statement
 selection-statement
 iteration-statement
 jump-statement

labeled-statement :=
 identifier : *statement*
 case *constant-expression* : *statement*
 default: *statement*

compound-statement :=
 { [*declaration-list*] [*statement-list*] }

declaration-list :=
 declaration
 declaration-list *declaration*

statement-list :=
 statement
 statement-list *statement*

expression-statement :=
 [*expression*];

selection-statement :=
 if (*expression*) *statement*

Syntax Summary

Phrase Structure Grammar

```
if ( expression ) statement else statement  
switch ( expression ) statement
```

```
iteration-statement :=  
while ( expression ) statement  
do statement while ( expression )  
for ( [expression]; [expression]; [expression] ) statement
```

```
jump-statement :=  
goto identifier ;  
continue ;  
break ;  
return [expression] ;
```

External Definitions

```
translation-unit :=  
external-declaration  
translation-unit external-declaration
```

```
external-declaration :=  
function-definition  
declaration
```

```
function-definition :=  
[declaration-specifiers] declarator [declaration-list]  
compound-statement
```

Preprocessing Directives

```
preprocessing-file :=  
    [group]  
  
group :=  
    group-part  
    group group-part  
  
group-part :=  
    [pp-tokens] new-line  
    if-section  
    control-line  
  
if-section :=  
    if-group [elif-groups] [else-group] endif-line  
  
if-group :=  
    # if      constant-expression new-line [group]  
    # ifdef   identifier new-line [group]  
    # ifndef  identifier new-line [group]  
  
elif-groups :=  
    elif-group  
    elif-groups elif-group  
  
elif-group :=  
    # elif    constant-expression new-line [group]  
  
else-group :=  
    # else    new-line [group]  
  
endif-group :=  
    # endif  new-line  
  
control-line :=  
    # include pp-tokens new-line  
    # define  identifier replacement-list new-line  
    # define  identifier( [identifier-list] ) replacement-list new-line  
    # undef   identifier new-line  
    # line    pp-tokens new-line  
    # error   [pp-tokens] new-line  
    # pragma  [pp-tokens] new-line  
    # new-line  
  
replacement-list :=  
    [pp-tokens]  
  
pp-tokens :=  
    preprocessing-token  
    pp-tokens preprocessing-token  
  
new-line :=  
    the new-line character
```

Syntax Summary
Preprocessing Directives

Index

Symbols

!, logical NOT operator, 114
#elif, 187
#else, 187
#endif, 187
#error, 192
#if, 187
#ifdef, 187
#ifndef, 187
#line, 190
#pragma, 191
%, modulo division operator, 82
%=, assignment operator, 86
&&, logical AND operator, 114
&, address of operator, 119
, dereferencing operator, 119
, multiplication operator, 82
*-, assignment operator, 86
++, increment operator, 110
+, addition operator, 82
+, unary plus operator, 82
+-, assignment operator, 86
+ESconstlit compiler option, 219
+help compiler option, 224
, comma operator, 104
-, subtraction operator, 82
--, decrement operator, 110
., structure and union member, 135
/, division operator, 82
/=: assignment operator, 86
<< =, assignment operator, 86
<<, shift left operator, 93
-=, assignment operator, 86
=: assignment operator, 86
 confused with equality operator (==), 128
==, equality operator
 confused with assignment operator (=), 128
>, greater than operator, 127
->, pointer member, 119

->, structure and union member, 135
>=, greater than or equal to operator, 127
>>, shift right operator, 93
>>=, assignment operator, 86
?., conditional operator, 105
__LP64__, 217
__VA_ARGS__, 267
_HPUX_SOURCE name space
 macro, 207
_PA_RISC2_0, 217
_POSIX_SOURCE name space
 macro, 207
_XOPEN_SOURCE name space
 macro, 207
||, logical OR operator, 114

A

a.out file, 200
address of operator, 119
aggregate
 initializing, 63
ALLOCS_NEW_MEMORY
 pragma, 247
ANSI C
 mode, 2, 198
 standard, 2
ANSI migration warnings, 226
arithmetic
 conversions, 78
arithmetic operators, 82
array, 80
 declarator, 57
 maximum number of dimensions, 262
 row-major storage, 57
 storage, 57
 subscripting, 84
assembly source files, 199
assignment
 conversion, 88, 276

 expression, 141
 operators, 86
auto keyword, 12
auto scalar objects, 63
auto storage class specifier, 38
AUTOPAGE pragma, 247, 281

B

Bextern compiler option, 208
bit operators, 93
bit-fields, 48, 78, 257
bitwise shift operators, 261
block of code, 148
braces in compound statements, 165
branch statements, 146
break statement, 146, 153

C

C libraries, 211
C9X macros, 267
calling functions, 107
 by reference, 109
 by value, 109
case keyword, 12
cast operator, 60, 99
casting
 integers to integers, 100
 pointers to pointers, 103
cc command, 198
CCOPTS environment variable, 233
char keyword, 12
char type, 40
character constants, 27
code offsets, 226, 284
comma operator (,), 104
comments in a source file, 9
compilation
 ANSI mode, 2
 compatibility mode, 3, 198
 compilation process, 198

Index

- compiler options, 207
- conditional, 178, 187
- PA-RISC architecture
 - versions, 235
- compiler options, 207
 - +DAmode, 215, 235
 - +DCaptype, 217
 - +DD32, 217
 - +DD64, 217
 - +DOosname, 218
 - +DSmodel, 217, 236
 - +e, 213
 - +ESconstlit, 219
 - +ESfsc, 219
 - +ESlit, 220
 - +ESnolit, 221
 - +ESnoparmreloc, 220
 - +ESsfc, 221
 - +f, 222
 - +FPflags, 222
 - +help, 224
 - +hugesize, 225
 - +k, 225
 - +L, 213, 226, 280
 - +M, 226
 - +m, 214, 226, 282
 - +M0, 226
 - +M1, 226
 - +M2, 226
 - +o, 214, 226
 - +Oopt, 227
 - +P, 227
 - +pa, 228
 - +pal, 228
 - +pgmname, 228
 - +r, 228
 - +sb, 229
 - +abytes, 229, 244
 - +uc, 229
 - +wn, 214, 229
 - +Z, 230
 - +z, 230
 - Aa, 207
 - Ac, 207
 - Bextern, 208
 - C, 280
 - Dname, 208
 - dynamic, 209
 - E, 209
 - fast, 209
 - G, 210
 - g, 210
 - Idir, 210
 - l, 211
 - Ldir, 211
 - N, 211
 - n, 211
 - noshared, 211
 - O, 212
 - o, 212
 - P, 212
 - p, 212
 - Q, 212
 - q, 212
 - S, 212
 - s, 212
 - summary, 200
 - t, 213
 - Uname, 213
 - V, 213
 - v, 213
 - w, 213
 - Wc, 214
 - Wc,-0, 282
 - Wc,-e, 213
 - Wc,-L, 213
 - Wc,-m, 214, 282
 - Wc,-o, 214, 284
 - Wc,-wn, 214
 - Wd, -a, 215
 - Wx, 213
 - Y, 215
 - y option, 215
 - Z, 215
 - z, 215
 - compound statement, 148
 - conditional compilation, 178, 187
 - conditional operator (?), 105
 - const keyword, 13
 - constant expressions, 139
 - constants, 24
 - integer, 24
 - continuation lines, 8
 - continue statement, 146, 154
 - conversions
 - arithmetic, 78
 - floating, 79
 - integral, 78
 - COPYRIGHT pragma, 241
 - COPYRIGHT_DATE pragma, 241
 - cpp (C preprocessor), 199, 208
 - CXperf profiler, 228

D

 - data alignment pragma, 242
 - data declarations, 262
 - data representation, 78
 - data type, 35
 - ranges, 254
 - sizes, 254
 - data types, 254
 - as implemented in HP C/HP-UX, 254
 - char, 40
 - double, 40
 - float, 40
 - int, 40
 - long, 40
 - long long, 40
 - short, 40
 - decimal constants, 24
 - declarations, 19, 35
 - declarator, 55
 - array, 57
 - compiling HP C programs, 198

Index

- pointer, 56
 - declaring variables, 19
 - decrement operators, 110
 - default keyword, 13, 171
 - default listing, 280
 - defined operator, 187
 - demand loadable, 212
 - dereferencing a pointer, 120
 - directives, preprocessor, 177
 - do/while statement, 156
 - dollar sign (\$) in identifier, 9
 - double keyword, 13
 - double type, 40
- E**
- ELF object modules, 216
 - else keyword, 13
 - enum declaration, 52
 - enumeration, 52
 - environment variables, 233
 - CCOPTS, TMPDIR, 233
 - TMPDIR, 234
 - error messages, 5
 - escape sequences, 27
 - ESnolit compiler option, 221
 - expressions, 81, 149
 - extern, 148
 - declarations, 262
 - keyword, 14
 - storage class specifier, 38
- F**
- fast compiler option, 209
 - FINI pragma, 239
 - float keyword, 14
 - float type, 40
 - floating-point
 - constants, 25
 - conversions, 79
 - expressions, 140
 - operations, 222
- G**
- types, 40
 - for statement, 158
 - function, 80, 265
 - call, 107
 - declarator, 58
 - definitions, 67
 - library, 195
 - prototypes, 67
 - referencing functions instead
 - of macros, 195
 - function calls
 - by reference, 109
 - by value, 109
- H**
- GNU standards, 267
 - goto statement, 146, 150, 162
 - gprof utility, 210
- I**
- header files
 - location, 195
 - specifying, 195
 - help, online, 5
 - hexadecimal constants, 24
 - HP C file location, 270
 - HP C source files, 199
 - HP specific type qualifier, 42, 270
 - HP_ALIGN pragma, 242, 243, 244
 - HP_DEFINED_EXTERNAL pragma, 250
 - HP_LONG_RETURN pragma, 250
 - HP_NO_RELOCATION pragma, 250
 - HP_SHLIB_VERSION pragma, 249
- J**
- identifier maps, 226, 282
- K**
- identifiers, 9
 - __VA_ARGS__, 267
 - case sensitivity, 10
 - IEEE floating-point format, 258
 - if statement, 152, 164
 - include files, 177
 - increment operators, 110
 - INIT pragma, 239
 - initialization
 - aggregates, 63
 - auto scalar objects, 63
 - expression, 158
 - static objects, 63
 - initialization of objects, 63
 - int keyword, 14
 - int type, 40
 - integer constants, 24
 - integers
 - expressions, 140
 - integral
 - conversions, 78
 - promotion, 74
 - unsigned preserving rules, 74
 - intrinsic, 276
 - defined, 276
 - INTRINSIC pragma, 240, 276
 - INTRINSIC_FILE pragma, 241, 278
- L**
- +L compiler option, 280
 - labeled statements, 150
 - language support, 215
 - lexical elements, 8
 - libraries, file location, 270
 - library functions, 195
-

Index

- line control, 190
- line number specification, 178
- LINES pragma, 246, 281
- linking, 198
- LIST pragma, 246, 281
- listing, 226
 - facility, 280
 - format, 280
 - options, 282
 - pragmas, 246, 281
- LOCALITY pragma, 241
- location of HP C files, 270
 - table, 270
- logical operators, 114, 116
- long keyword, 14
- long long type, 40
- long type, 40
- looping statements, 151
- lvalue, 141
 - expressions, 141
- M**
- +m compiler option, 282
- macro, 265
 - definition, 182
 - predefined, 186
 - replacement, 178, 182
- macros
 - C9X, 267
 - variable argument, 267
 - variadic, 267
- math libraries, 264
 - location of files, 272
- maximum number of array dimensions, 262
- mismatched ABI error, 198
- multi-character constant, 28
- N**
- name spaces, 22, 207
- names
 - type, 60
- Native Language Support (NLS), 215
- nesting level, 226
- newline characters, 8
- compiler options
 - +, 227
- NO_SIDE_EFFECTS pragma, 248
- [NO]INLINE pragma, 248
- no-operation statements, 149
- [NO]PTRS_STRONGLY_TYPE D pragma, 248
- null statement, 149
- O**
- objdebug, 227
- octal constants, 24
- online help, 5
- operator, 81
 - arithmetic, 82
 - assignment, 86
 - bit, 93
 - cast, 99
 - comma (,), 104
 - conditional (?:), 105
 - decrement, 110
 - defined, 187
 - increment, 110
 - logical, 114
 - precedence, 136
 - sizeof, 133
- optimization, 198, 212, 227
- OPTIMIZE pragma, 247
- overflow expression, 261
- P**
- PACK pragma, 242, 245
- PAGE pragma, 246, 281
- parallel runtime library, 273
- PA-RISC architecture versions, 235
- pointer, 80, 261
 - assigning address value, 119
 - casting pointers to pointers, 103
 - declarator, 56
 - dereferencing, 120
 - expressions, 141
 - portability, 1
 - pragmas, 177, 178, 238
 - ALLOCS_NEW_MEMORY, 247
 - AUTOPAGE, 247, 281
 - COPYRIGHT, 241
 - COPYRIGHT_DATE, 241
 - data alignment, 242
 - fast libcall, 250
 - FINI, 239
 - HP_ALIGN
 - DOMAIN_NATURAL, 243
 - HP_ALIGN
 - DOMAIN_WORD, 243
 - HP_ALIGN
 - HPUX_NATURAL, 243
 - HP_ALIGN
 - HPUX_NATURAL_S500, 243
 - HP_ALIGN HPUX_WORD, 243
 - HP_ALIGN NATURAL, 243
 - HP_DEFINED_EXTERNAL, 250
 - HP_LONG_RETURN, 250
 - HP_NO_RELOCATION, 250
 - HP_SHLIB_VERSION, 249
 - INIT, 239
 - initialization and termination, 238
 - INTRINSIC, 240, 276
 - INTRINSIC_FILE, 241, 278
 - LINES, 246, 281
 - LIST, 246, 281
 - LOCALITY, 241
 - NO_SIDE_EFFECTS, 248
 - [NO]INLINE, 248

Index

- [NO]PTRS_STRONGLY_TYPED, 248
- OPTIMIZE, 247
- PACK, 245
- PAGE, 246, 281
- SUBTITLE, 246, 281
- TITLE, 246, 281
- VERSIONID, 242
- WIDTH, 246, 281
- precedence of operators, 136
- predefined macros
 - DATE, 186
 - FILE, 186
 - LINE, 186
 - STDC, 186
 - TIME, 186
- preprocessing directives
 - #define, 182
 - #elif, 187
 - #else, 187
 - #endif, 187
 - #error, 192
 - #if, 187
 - #ifdef, 187
 - #ifndef, 187
 - #include, 180
 - #line, 190
 - #pragma, 191
- preprocessor, 198, 208, 212
- profile-based optimization, 227
- profiling, 210
- promoting floating-point numbers as floats, 222
- promotion, integral, 74
- Q**
- Quiet Changes, 226
- R**
- ranges of data types, 254
- register keyword, 15
- register storage class specifier, 38
- relational expressions
 - evaluation, 129
- relational operators, 127
- relocatable object file, 34, 199
- return statement, 146, 168
- row-major array storage, 57
- running HP C programs, 251
- S**
- scalar objects
 - initializing, 63
- scope, 262
- selection statements, 152
- self-referential structure, 50
- shared libraries, 198
- shared library pragma, 248
- short keyword, 15
- short type, 40
- signed keyword, 15
- simple assignment, 86
- sized enum, 53
- sizeof operator, 60, 133
- sizes of data types, 254
- source files
 - assembly, 199
 - HP C, 199
 - inclusion, 180
 - parts of C program, 8
- specifier
 - storage-class, 38
 - structure, 47
 - type, 40
 - union, 47
- statement
 - branching, 146
 - break, 146, 153
 - compound, 148
 - continue, 146, 154
 - defined, 145
 - do/while, 156
 - for, 158
 - goto, 146, 162
 - groups, 145
 - if, 164
 - labeled, 150
 - looping, 151
 - no-operation, 149
 - null, 149
 - return, 146, 168
 - selection, 152
 - switch, 171
 - use of semicolon, 145
 - while, 175
- static keyword, 16
- static objects, 63
- static storage class specifier, 38
- static variable, 148
- stdarg.h header file, 265
- storage-class specifiers, 38
- string constants, 29
- structure, 47, 260
 - members, 135
 - self-referential, 50
 - specifier, 47
 - tag, 49
- structure and union member (.), 135
- structure and union pointer (arrow), 135
- subprocess, 213
- SUBTITLE pragma, 246, 281
- switch statement, 150, 152, 171
- symbol table, 212
- symbolic debugger, 210
- T**
- tag
 - structure, 49
 - union, 49
- thread HP specific type qualifier, 42
- thread keyword, 16

Index

TITLE pragma, 246, 281
TMPDIR environment variable, 234
translation unit, 34
trigraph sequences, 193
type
 const qualifier, 44
 definitions, 61
 enumeration, 52
 mismatches in external names, 261
 names, 60
 specifiers, 40
 volatile qualifier, 44
typedef declarations, 19
typedef keyword, 38, 61

U

underflow expression, 261
underscore (_) in identifier, 9
union, 48, 260
 members, 135
 specifier, 47
 tag, 49
unsigned keyword, 17, 40
/usr/lib/sysintr, 278

V

va_args macros, 265
va_dcl macro, 265
va_start macro, 265
value preserving rules, 74
variables
 syntax for declaring, 19
verbose mode, 213
version print, 213
VERSIONID pragma, 242
void type, 40
volatile type qualifier, 18, 44

W

wchar_t typedef, 65

while statement, 175
white space, 8
wide char constant, 28
WIDTH pragma, 246, 281