HP 9000 Computer Systems

HP C/HP-UX Reference Manual

Workstations and Servers

HP Part No. 92453-90085
Printed in U.S.A. May 1997

E0597

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

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information that 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.

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

HEWLETT-PACKARD COMPANY
3000 Hanover Street
Palo Alto, California 94304 U.S.A.

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

© Copyright 1989, 1991, 1992, 1993, 1994, 1996, 1997 by
HEWLETT-PACKARD COMPANY

Printing History

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

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

First Edition	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

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

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

Preface

This manual presents reference information on the C programming language, as implemented on HP 9000 computers. It presents information specific to writing and executing C programs on the HP-UX operating system. This manual is intended for experienced C programmers who are familiar with HP computer systems.

Manual Organization

This manual is organized as follows:

Chapter 1 **Introduction**
Provides an introduction to the HP C programming language.

Chapter 2 **Lexical Elements**
Presents the lexical elements of HP C, including tokens, keywords, identifiers, constants, punctuation, and comments.

Chapter 3 **Data Types and Declarations**
Describes C data types, declarations, type specifiers, storage-class specifiers, structure and union specifiers, enumerations, type names, and initialization.

Chapter 4 **Type Conversions**
Explains type conversions that occur when different data types are used within a program.

Chapter 5 **Expressions**
Describes how to form expressions in HP C and includes information on operators and operator precedence.

Chapter 6 **Statements**
Provides details on HP C statements.

Chapter 7 **Preprocessing Directives**
Describes preprocessor directives that function as compiler control lines.

Chapter 8 **C Library Functions**
Lists the header files that define the objects found in each library and describes how to use library functions.

Chapter 9 **Compiling and Running HP C Programs**
Describes how to compile and run an HP C program on the HP-UX operating system.

Chapter 10 **HP C/HP-UX Implementation Topics**
Presents information specific to programming in C on HP 9000 computers.

Chapter 11 **Using Intrinsics**
Describes how to call the external routines known as *intrinsics*.

Chapter 12 **The Listing Facility**
Explains the listing format of the HP C compiler and describes facilities that can be used to define characteristics of the format.

Appendix A **Syntax Summary**
Summarizes the HP C language syntax.

Related Information

Refer to the following materials for further information on C programming.

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

HP C Programmer's Guide—This programming guide explains how to program in HP C and gives detailed descriptions of storage and alignment, the optimizer, HP C debugging, and programming for efficiency and portability.

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-UX Reference—For HP-UX 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.

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

Conventions

This manual uses a variation of the Backus-Naur form to describe the HP C language. The language is described in terms of syntactic categories (nonterminals). Syntax descriptions define the syntactic categories. The ::= symbol following a syntactic category introduces its definition. Alternate definitions are listed on separate lines unless preceded by "one of the following" or an equivalent expression.

A definition of a syntactic category can be recursive. For example,

expression ::=
assignment-expression
expression, assignment-expression

The second alternate definition for *expression* contains *expression*. This allows for *expression* to consist of any number of *assignment-expressions*, separated by commas.

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

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: `break;`
italics	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 *identifier* with the name of a label you want the program to transfer execution to at this point: `goto` *identifier*`;`
[ ]	Within syntax descriptions, italicized brackets surround optional elements. For example, the *expression* in the `return` statement is optional: `return` *[expression]*`;`
⋮	Within examples, vertical ellipses may show where portions of the example were omitted.

Contents

1. Introduction

- ANSI Mode 1-2
- Compatibility Mode 1-2
- Focus of this Manual 1-3
- HP C Online Help 1-3
  - Accessing HP C Help with the `+help` Option 1-3
  - Accessing HP C Help with the Front Panel 1-3
  - Accessing HP C Help with the dthelpview Command 1-4

2. Lexical Elements

- Tokens 2-1
- Keywords 2-2
- Identifiers 2-3
- Constants 2-12
- Floating Constants 2-12
- Integer Constants 2-14
- Enumeration Constants 2-18
- Character Constants 2-18
- String Literals 2-22
- Operators 2-24
- Punctuators 2-25
- Comments 2-26

3. Data Types and Declarations

- Program Structure 3-2
- Declarations 3-3
- Storage-Class Specifiers 3-5
- Type Specifiers 3-7
- HP Specific Type Qualifiers 3-9
- Type Qualifiers 3-11

Structure and Union Specifiers 3-14
Enumeration 3-20
Declarators 3-23
Type Names 3-28
Type Definitions Using typedef 3-30
Initialization 3-32
Function Definitions 3-37
Four-Byte Extended UNIX Code (EUC) 3-41

4. Type Conversions
Integral Promotions 4-2
Usual Arithmetic Conversions 4-3
Arithmetic Conversions 4-5
Integral Conversions 4-5
Floating Conversions 4-6
Arrays, Pointers, and Functions 4-7

5. Expressions
Operator Precedence 5-2
Lvalue Expressions 5-4
Primary Expressions 5-5
Postfix Operators 5-6
Array Subscripting 5-7
Function Calls 5-9
Structure and Union Members 5-12
Postfix Increment and Decrement Operators 5-13
Unary Operators 5-14
Prefix Increment and Decrement Operators 5-15
Address and Indirection Operators 5-16
Unary Arithmetic Operators 5-17
The sizeof Operator 5-18
Cast Operators 5-19
Multiplicative Operators 5-20
Additive Operators 5-22
Bitwise Shift Operators 5-24
Relational Operators 5-25
Equality Operators 5-27
Bitwise AND Operator 5-29

Bitwise Exclusive OR Operator 5-30
Bitwise Inclusive OR Operator 5-31
Logical AND Operator 5-32
Logical OR Operator 5-33
Conditional Operator 5-34
Assignment Operators 5-36
Comma Operator 5-39
Constant Expressions 5-40

6. Statements
Labeled Statements 6-2
Compound Statement or Block 6-3
Expression and Null Statements 6-5
Selection Statements 6-6
The if Statement 6-7
The switch Statement 6-9
Iteration Statements 6-11
The while Statement 6-13
The do Statement 6-14
The for Statement 6-15
Jump Statements 6-17
The goto Statement 6-19
The continue Statement 6-20
The break Statement 6-21
The return Statement 6-22

7. Preprocessing Directives
Source File Inclusion 7-4
Macro Replacement 7-6
Predefined Macros 7-10
Conditional Compilation 7-11
Line Control 7-15
Pragma Directive 7-16
Error Directive 7-17
Trigraph Sequences 7-18

8. C Library Functions

9. Compiling and Running HP C Programs
Compiling HP C Programs 9-1
Compatibility Mode vs. ANSI C Mode 9-2
The cc(1) Command 9-2
Specifying Files to the cc Command 9-2
Specifying Options to the cc Command 9-3
An Example of Using a Compiler Option 9-3
Concatenating Options 9-3
HP C Compiler Options 9-4
Examples of Compiler Commands 9-27
Environment Variables 9-29
CCOPTS Environment Variable 9-29
TMPDIR Environment Variable 9-30
Compiling for Different Versions of the PA-RISC Architecture . 9-30
Using +DA to Generate Code for a Specific Version of PA-RISC 9-30
Using +DS to Specify Instruction Scheduling 9-31
Compiling in Networked Environments 9-32
Pragmas 9-33
Intrinsic Pragmas 9-33
INTRINSIC Pragma 9-33
INTRINSIC_FILE Pragma 9-33
Copyright Notice and Identification Pragmas 9-34
COPYRIGHT Pragma 9-34
COPYRIGHT_DATE Pragma 9-34
LOCALITY Pragma 9-34
VERSIONID Pragma 9-35
Optimization Pragmas 9-35
ALLOCS_NEW_MEMORY Pragma 9-35
FLOAT_TRAPS_ON Pragma 9-35
[NO]INLINE Pragma 9-35
[NO]PTRS_STRONGLY_TYPED Pragma 9-36
NO_SIDE_EFFECTS Pragma 9-36
Shared Library Pragma 9-36
HP_SHLIB_VERSION Pragma 9-36
Data Alignment Pragma 9-37

HP_ALIGN Pragma 9-37
Data Alignment Stack 9-37
Alignment Modes 9-38
Accessing Data with the HP_ALIGN Pragma 9-39
Listing Pragmas 9-41
LINES Pragma 9-41
WIDTH Pragma 9-41
TITLE Pragma 9-42
SUBTITLE Pragma 9-42
PAGE Pragma 9-42
LIST Pragma 9-42
AUTOPAGE Pragma 9-42
Running HP C Programs 9-43

10. HP C/HP-UX Implementation Topics

Data Types 10-1
Bit-Fields 10-3
IEEE Floating-Point Format 10-4
Lexical Elements 10-6
Structures and Unions 10-6
Type Mismatches in External Names 10-7
Expressions 10-7
Pointers 10-7
Maximum Number of Dimensions of an Array 10-8
Scope of extern Declarations 10-8
Conversions Between Floats, Doubles, and Long Doubles 10-8
Statements 10-9
Preprocessor 10-9
Library Functions and Header Files 10-9
The Math Library 10-9
Other Library Functions 10-10
The varargs Macros 10-10
Example 10-11
HP Specific Type Qualifiers 10-13
Location of Files 10-14

11. Using Intrinsics

INTRINSIC Pragma 11-2
  Examples 11-2
INTRINSIC_FILE Pragma 11-4

12. The Listing Facility

Listing Format 12-1
  Compatibility Mode 12-1
  ANSI Mode 12-1
Listing Pragmas 12-2
Listing Options 12-3
  Identifier Maps 12-3
  Code Offsets 12-7
  Example 12-7

A. Syntax Summary

Lexical Grammar A-1
  Tokens A-1
  Keywords A-2
  Identifiers A-2
  Constants A-3
  String Literals A-6
  Operators A-6
  Punctuators A-7
  Header Names A-7
  Preprocessing Numbers A-8
Phrase Structure Grammar A-9
  Expressions A-9
  Declarations A-12
  Statements A-16
  External Definitions A-17
Preprocessing Directives A-18

Index

Figures

2-1. C Types 2-11
10-1. Internal Representation of Floating-Point Numbers 10-4

Tables

2-1. Special Characters 2-21
3-1. C Type Specifiers 3-8
3-2. Declarations using const and volatile 3-12
5-1. C Operator Precedence 5-3
7-1. Predefined Macros 7-10
7-2. Trigraph Sequences and Replacement Characters 7-18
9-1. HP C Compiler Options at a Glance 9-5
9-2. HP C Compiler Option Details 9-9
10-1. HP C/HP-UX Data Types 10-2
10-2. Location of Files 10-14

1

Introduction

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.

Work on a standard for C began in 1983. The *Draft Proposed American National Standard for Information Systems--Programming Language C* was completed and was approved by the Technical Committee X3J11 on the C Programming Language in September, 1988. It was forwarded to X3, the American National Standards Committee on Computers and Information Processing, early in 1989. It became an American National Standard in December, 1989.

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.

The creation of an ANSI standard for C raises the question of compatibility with preexisting implementations of the language. For the most part, the committee that developed the standard adopted the goal of codifying existing practice, rather than introducing new language features that had never been tried. They went to great lengths to minimize changes which would "break" existing programs.

Many programs will compile and execute properly in an ANSI C environment with no changes. In the vast majority of cases where a change is required, the offending construct will be identified by a warning or error message produced

by the compiler. In a few cases, which are believed to be rare in actual practice, certain program constructs will be accepted but will behave differently under ANSI C. HP C/HP-UX is capable of producing migration warnings to help identify code where such “quiet changes” would occur.

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.

At the HP-UX 10.20 operating system release, compatibility mode was the default.

Focus of this Manual

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. The online help can be accessed from any X Windows display device. Several methods of invoking the HP C online help are listed below.

Note that error messages are documented in the online help.

Accessing HP C Help with the `+help` Option

You may access HP C online help with the command line:

```
cc +help
```

Accessing HP C Help with the Front Panel

To access HP C online help if HP C and the help system are installed on your workstation:

1. Click on the `?` icon on the HP VUE front panel.
2. The "Welcome to Help Manager" menu appears. Click on the HP C icon.

Accessing HP C Help with the dthelpview Command

If HP C is installed on another system or you are not running the help system, enter the following command from the system where HP C is installed:

```
/usr/dt/bin/dthelpview -h c
```

2

Lexical Elements

This chapter describes the lexical elements of the C language, using Backus-Naur form.

Tokens

A *token* is the smallest lexical element of the C language.

Syntax

token `::=` *keyword*
identifier
constant
string-literal
operator
punctuator

Description

The compiler combines input characters together to form the longest token possible when collecting characters into tokens. For example, the sequence `integer` is interpreted as a single identifier rather than the reserved keyword `int` followed by the identifier `eger`.

A token cannot exceed 509 characters in length. Consecutive source code lines can be concatenated together using the backslash (\) character at the end of the line to be continued. The total number of characters in the concatenated source lines cannot exceed 509.

Tokens

The term *white space* refers to the set of characters that includes spaces, horizontal tabs, newline characters, vertical tabs, form feeds, and comments. You can use white space freely between tokens, and extra spaces are ignored in your programs. But note that at least one space may be required to separate tokens. So, a character such as a hyphen (-) can take on different meanings depending upon the white space around it.

For example:

```
a- -1
```

is different from

```
a--1
```

Keywords

The following keywords are reserved in the C language. You cannot use them as program identifiers. Type them as shown, using lowercase characters.

```
auto      do      goto     signed       union
break     double  if       sizeof       unsigned
case      else    int      static       void
char      enum    long     struct       volatile
const     extern  register switch       while
continue  float   return   __thread (HP-UX 10.30 and later)
default   for     short    typedef
```

Identifiers

An *identifier* is a sequence of characters that represents an entity such as a function or a data object.

Syntax

```
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
```

Description

An identifier must start with a nonnumeric character followed by a sequence of digits or nonnumeric characters. Internal and external names may have up to 255 significant characters.

Identifiers are case sensitive. The compiler considers upper- and lowercase characters to be different. For example, the identifier `CAT` is different from the identifier `cat`. This is true for external as well as internal names.

An HP extension to the language in compatibility mode allows `$` as a valid character in an identifier as long as it is not the first character.

The following are examples of legal and illegal identifiers:

Identifiers

Legal Identifiers

```
Sub_Total
X
aBc
Else
do_123
```

Illegal Identifiers

`3xyz`	First character is a digit.
`const`	Conflict with a reserved word.
`#note`	First character not alphabetic or _.
`Num'2`	Contains an illegal character.

All identifiers that begin with the underscore (_) character are reserved for system use. If you define identifiers that begin with an underscore, the compiler may interpret them as internal system names. The resulting behavior is undefined.

Finally, identifiers cannot have the same spelling as reserved words. For example, `int` cannot be used as an identifier because it is a reserved word. `INT` is a valid identifier because it has different case letters.

Identifier Scope

The scope of an identifier is the region of the program in which the identifier has meaning. There are four kinds of scope:

1. **File Scope**—Identifiers declared outside of any block or list of parameters have scope from their declaration point until the end of the translation unit.
2. **Function Prototype Scope**—If the identifier is part of the parameter list in a function declaration, then it is visible only inside the function declarator. This scope ends with the function prototype.
3. **Block Scope**—Identifiers declared inside a block or in the list of parameter declarations in a function definition have scope from their declaration point until the end of the associated block.
4. **Function Scope**—Statement labels have scope over the entire function in which they are defined. Labels cannot be referenced outside of the function in which they are defined. Labels do not follow the block scope rules. In

particular, `goto` statements can reference labels that are defined inside iteration statements. Label names must be unique within a function.

A preprocessor macro is visible from the `#define` directive that declares it until either the end of the translation unit or an `#undef` directive that undefines the macro.

Identifier Linkage

An identifier is *bound* to a physical object by the context of its use. The same identifier can be bound to several different objects at different places in the same program. This apparent ambiguity is resolved through the use of scope and name spaces. The term *name spaces* refers to various categories of identifiers in C (see "Name Spaces" later in this chapter for more information).

Similarly, an identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*. There are three kinds of linkage:

1. **Internal**—within a single translation unit, each instance of an identifier with internal linkage denotes the same object or function.
2. **External**—within all the translation units and libraries that constitute an entire program, each instance of a particular identifier with external linkage denotes the same object or function.
3. **None**—identifiers with no linkage denote unique entities.

If an identifier is declared at file scope using the storage-class specifier `static`, it has internal linkage.

If an identifier is declared using the storage-class specifier `extern`, it has the same linkage as any visible declaration of the identifier with file scope. If there is no visible declaration with file scope, the identifier has external linkage.

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 storage-class specifier `extern`. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

The following identifiers have no linkage: an identifier declared to be anything other that an object or a function; an identifier declared to be a function

parameter; and a block scope identifier for an object declared without the storage-class specifier `extern`.

For example:

```
extern int i;             /* External linkage */
static float f;           /* Internal linkage */
struct Q { int z; };      /* Q and z both have no linkage */

static int func()         /* Internal linkage */
{
   extern int temp;       /* External linkage */
   static char c;         /* No linkage */
   int j;                 /* No linkage */
   extern float f;        /* Internal linkage; refers to */
                          /* float f at file scope */
 }
```

Two identifiers that have the same scope and share the same name space cannot be spelled the same way. Two identifiers that are not in the same scope or same name space can have the same spelling and will bind to two different physical objects. For example, a formal parameter to a function may have the same name as a structure tag in the same function. This is because the two identifiers are not in the same name space.

If one identifier is defined in a block and another is defined in a nested (subordinate) block, both can have the same spelling.

For example:

```
{
   int i;        <---A
     .
     .           <---B
     .
   {
      float i;   <---C
          .      <---D
          .
          .
```

```
        }               <---E
        .
        .               <---F
        .
    }                   <---G
```

In the example above, the identifier `i` is bound to two physically different objects. One object is an integer and the other is a floating-point number. Both objects, in this case, have block scope. At location `A,` identifier `i` is declared. Its scope continues until the end of the block in which it is defined (point `G`). References to `i` at location `B` refer to an integer object.

At point `C,` another identifier is declared. The previous declaration for `i` is *hidden* by the new declaration until the end of the block in which the new `i` is declared. References to the identifier `i` result in references to a floating-point number (point `D`). At the end of the second block (point `E`), the floating-point declaration of `i` ends. The previous declaration of `i` again becomes visible, and references to identifier `i` at point `F` reference an `int`.

Storage Duration

Identifiers that represent variables have a real existence at run time, unlike identifiers that represent abstractions like `typedef` names or structure tags. The duration of an object's existence is the period of time in which the object has storage allocated for it. There are two different durations for C objects:

1. **Static**—An object whose identifier is declared with external or internal linkage, or with the storage-class specifier `static`, has static storage duration. Objects with static storage duration have storage allocated to them when the program begins execution. The storage remains allocated until the program terminates.
2. **Automatic**—An object whose identifier is declared with no linkage, and without the storage-class specifier `static`, has automatic storage duration. Objects with automatic storage duration are allocated when entering a function and deallocated on exit from a function. If you do not explicitly initialize such an object, its contents when allocated will be indeterminate. Further, if a block that declares an initialized automatic duration object is not entered through the top of the block, the object will not be initialized.

Name Spaces

In any given scope, you can use an identifier for only one purpose. An exception to this rule is caused by separate name spaces. Different name spaces allow the same identifier to be *overloaded* within the same scope. This is to say that, in some cases, the compiler can determine from the context of use which identifier is being referred to. For example, an identifier can be both a variable name and a structure tag.

Four different name spaces are used in C:

1. **Labels**—The definition of a label is always followed by a colon (:). A label is only referenced as the object of a `goto` statement. Labels, therefore, can have the same spelling as any nonlabel identifier.
2. **Tags**—Tags are part of structure, union, and enumeration declarations. All tags for these constructs share the same name space (even though a preceding `struct, union` or `enum` keyword could clarify their use). Tags can have the same spelling as any non-tag identifier.
3. **Members**—Each structure or union has its own name space for members. Two different structures can have members with exactly the same names. Members are therefore tightly bound to their defining structure. For example, a pointer to structure of type A cannot reference members from a structure of type B. (You may use unions or a cast to accomplish this.)
4. **Other names**—All other names are in the same name space, including variables, functions, `typedef` names, and enumeration constants.

Conceptually, the macro prepass occurs before the compilation of the translation unit. As a result, macro names are independent from all other names. Use of macro names as ordinary identifiers can cause unwanted substitutions.

Types

The *type* of an identifier defines how the identifier can be used. The type defines a set of values and operations that can be performed on these values. There are three major categories of types in C—object type, function type, and incomplete type.

I. **Object Type**

There are 3 object types—scalar, aggregate, and union. These are further subdivided (see figure 2-1).

1. **Scalar**—These types are all objects that the computer can directly manipulate. Scalar types include pointers, numeric objects, and enumeration types.

   a. **Pointer**—These types include pointers to objects and functions.

   b. **Arithmetic**—These types include floating and integral types.

      i. **Floating**: The floating types include the following:

         float—A 32-bit floating point number.
         double—A 64-bit double precision floating point number.
         long double—A 128-bit quad precision floating point number.

      ii. **Integral**: The integral types include all of the integer types that the computer supports. This includes type `char`, signed and unsigned integer types, and the enumerated types.

         char—An object of `char` type is one that is large enough to store an ASCII character. Internally, a `char` is a signed integer.

         Integer—Integers can be `short`, `long`, `int`, or `long long`; they are normally signed, but can be made unsigned by using the keyword `unsigned` with the type. In C, a computation involving unsigned operands can never overflow; high-order bits that do not fit in the result field are simply discarded without warning. A `short int` is a 16-bit integer. The `int` and `long int` integers are 32-bit integers. A `long long int` is a 64-bit integer. Integer types include `signed char` and `unsigned char` (but not "plain" `char`).

         Enumerated—Enumerated types are explicitly listed by the programmer; they name specified integer constant values. The enumerated type `color` might, for example, define `red, blue`, and `green.` An object of type `enum color` could then have the value `red, blue,` or `green.` As an extension to the HP C compiler, it is possible to override the default allocation of four bytes for enumerated variables by specifying a type in the declaration. For

example, a `short enum` is two bytes long and a `char enum` is one byte.

2. **Aggregate**—Aggregate types are types that are composed of other types. With some restrictions, aggregate types can be composed of members of all of the other types including (recursively) aggregate types. Aggregate types include:

   a. **Structures**—Structures are collections of heterogeneous objects. They are similar to Pascal records and are useful for defining special-purpose data types.

   b. **Arrays**—Arrays are collections of homogeneous objects. C arrays can be multidimensional with conceptually no limit on the number of dimensions.

3. **Unions**—Unions, like structures, can hold different types of objects. However, all members of a union are "overlaid"; that is, they begin at the same location in memory. This means that the union can contain only one of the objects at any given time. Unions are useful for manipulating a variety of data within the same memory location.

II. **Function Type**

A function type specifies the type of the object that a function returns. A function that returns an object of type `T` can be referred to as a "function returning `T`", or simply, a `T` function.

III. **Incomplete Type**

The `void` type is an incomplete type. It comprises an empty set of values. Only pointers and functions can have void type. A function that returns void is a function that returns nothing. A pointer to void establishes a generic pointer.

Figure 2-1 illustrates the C types.

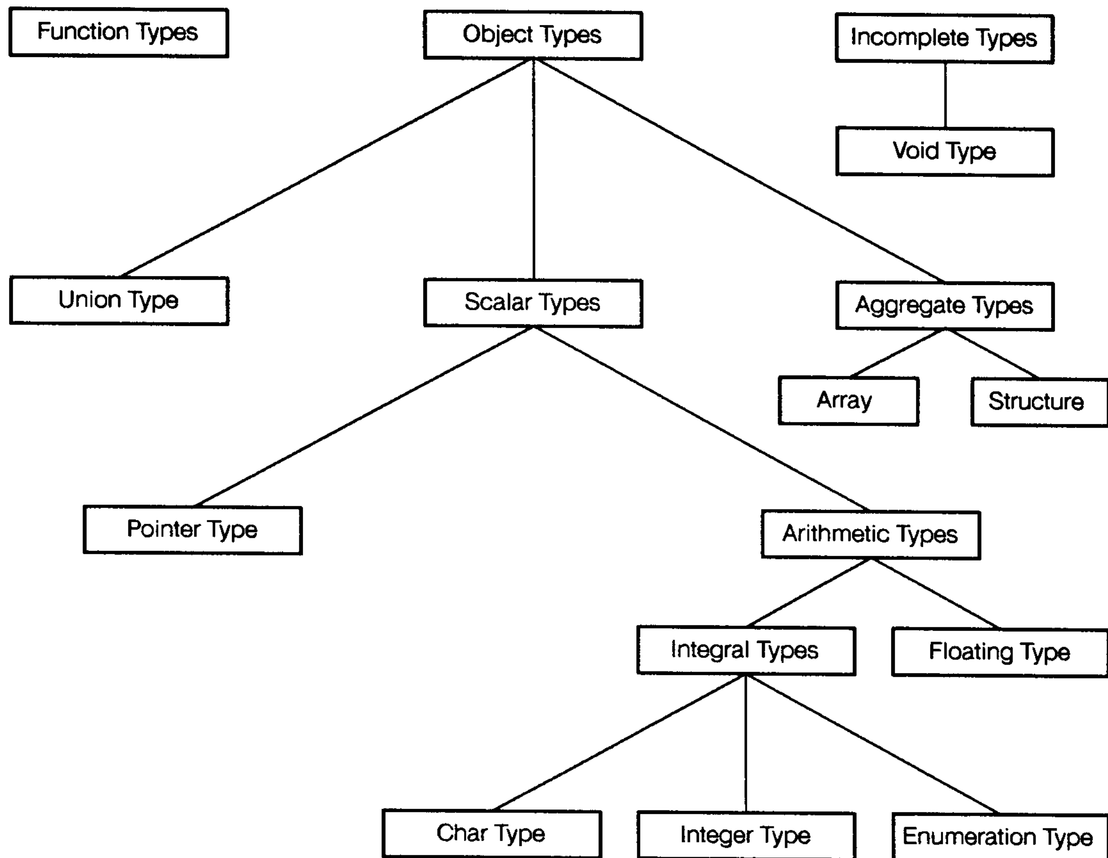

Figure 2-1. C Types

Constants

A *constant* is a primary expression whose literal or symbolic value does not change.

Syntax

```
constant ::= floating-constant
             integer-constant
             enumeration-constant
             character-constant
```

Description

Each constant has a value and a type. Both attributes are determined from its form. Constants are evaluated at compile time whenever possible. This means that expressions such as

```
2+8/2
```

are automatically interpreted as a single constant at compile time.

Floating Constants

Floating constants represent floating-point values.

Syntax

```
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 `::=`
    `+`
    `-`

digit-sequence `::=`
    digit
    digit-sequence digit

floating-suffix `::=`
    `F`
    `f`
    `L`
    `l`

Note Suffixes in floating-constants are available only in ANSI mode.

Description

A floating constant has a *value part* that may be followed by an *exponent part* and a suffix specifying its type. The value part may include a digit sequence representing the whole-number part, followed by a period (.), followed by a digit sequence representing the fraction part. The exponent includes an `e` or an `E` followed by an exponent consisting of an optionally signed digit sequence. Either the whole-number part or the fraction part must be used; either the period or the exponent part must be used.

The format of floating-point numbers is given in Chapter 10, "HP C/HP-UX Implementation Topics."

A floating constant may include a suffix that specifies its type. `F` or `f` specifies type `float` (single precision). `L` or `l` specifies `long double` (quad precision). The default type (unsuffixed) is `double`.

Examples

`3.28e+3f`	*float constant = 3280*
`6.E2F`	*float constant = 600*
`201e1L`	*long double constant = 2010*
`4.8`	*double constant = 4.8*

Integer Constants

Integer constants represent integer values.

Syntax

integer-constant `::=`
    decimal-constant [integer-suffix]
    octal-constant [integer-suffix]
    hex-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 [length-suffix]
length-suffix [unsigned-suffix]

unsigned-suffix `::=` `any character from the set`

```
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
```

Note The `u` and `U` suffixes are available only in ANSI mode (`-Aa` option).

Note The `ll`, `LL`, `Ll`, and `lL` suffix is not available under strict ANSI (`-Aa`) compilation mode.

Description

An integer constant begins with a digit, but has no period or exponent part. It may have a prefix that specifies its base (decimal, octal, or hexadecimal) and suffix that specifies its type.

The size and type of integer constants are described in Chapter 10, "HP C/HP-UX Implementation Topics."

Octal constants begin with a zero and can contain only octal digits. Several examples of octal constants are:

```
077   01L   01234567 02221
```

Hexadecimal constants begin with either `0x` or `0X`. The case of the `x` character makes no difference to the constant's value. The following are examples of hexadecimal constants:

```
0xACE   0XbAf  0x12L
```

The suffix `L` or `l` stands for long. The suffix `ll`, `LL`, `lL`, or `Ll` stands for long long. The suffix `U` or `u` stands for unsigned. These suffixes can be used on all three types of integer constants (decimal, octal, and hexadecimal).

The type of an integer constant is the first of the corresponding list in which its value can be represented, as shown:

- Unsuffixed decimal: `int, unsigned long int`.
- Unsuffixed octal or hexadecimal: `int, unsigned int`.
- Suffixed by the letter `u` or `U`: `unsigned int`.
- Suffixed by the letter `l` or `L`: `long int, unsigned long int`.
- Suffixed by both the letters `u` or `U` and `l` or `L`: `unsigned long int`.
- Suffixed by the letters `ll`, `LL`, `Ll`, or `lL`: `long long int, unsigned long long int`.
- Suffixed by both the letters `u` or `U` and `ll`, `LL`, `Ll`, or `lL`: `unsigned long long int`.

Examples

`0xFFFFu`	*unsigned hexadecimal integer*
`4196L`	*signed long decimal integer*
`0X89ab`	*signed hexadecimal integer*
`047L`	*signed long octal integer*
`64U`	*unsigned decimal integer*
`15`	*signed 32-bit decimal integer*
`15L`	*signed long (32-bit) decimal integer*
`15LL`	*signed 64-bit decimal integer*
`15U`	*unsigned decimal integer*
`15UL`	*unsigned long decimal integer*
`15LU`	*unsigned long decimal integer*
`15ULL`	*unsigned long long decimal integer*
`0125`	*signed 32-bit octal integer*
`0125ll`	*signed 64-bit octal integer*

Enumeration Constants

Enumeration constants are identifiers defined to have an ordered set of integer values.

Syntax

enumeration-constant `::=` *identifier*

Description

The identifier must be defined as an enumerator in an `enum` definition. Enumeration constants are specified when the type is defined. An enumeration constant has type `int`.

Character Constants

A *character constant* is a constant that is enclosed in single quotes.

Syntax

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 ::= one of
        \'  \"  \?  \\
        \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
```

Note `\a` and `\?` are available only in ANSI mode.

Description

There are two types of character constants—*integral character constants* and *wide character constants*.

Integral character constants are of type `int`. They do not have type `char`. However, because a `char` is normally converted to an `int` in an expression, this seldom is a problem. The contents can be ASCII characters, octal escape sequences, or hexadecimal escape sequences.

Octal escape sequences consist of a backslash, (\ ) followed by up to three octal digits. Hexadecimal escape sequences also start with a backslash, which is followed by lowercase x and any number of hexadecimal digits. It is terminated by any non-hexadecimal characters.

The digits of the escape sequences are converted into a single 8-bit character and stored in the character constant at that point. For example, the following character constants have the same value:

```
'A'          '\101'        '\x41'
```

They all represent the decimal value 65.

Character constants are not restricted to one character; multi-character constants are allowed. 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.

Wide character constants (type `wchar_t`) are of type `unsigned int`. A wide character constant is a sequence of one or more multibyte characters enclosed in single quotes and prefixed by the letter `L`. The value of a wide character constant containing a single multibyte character is a member of the extended execution character set whose value corresponds to that of the multibyte character. The value of a multibyte character can be found by calling the function `mbtowc`.

For multi-character wide character constants, the entire content of the constant is extracted into an unsigned integer and the resulting character is represented by the final value.

Some characters are given special representation in escape sequences. These are nonprinting and special characters that programmers often need to use (listed in Table 2-1).

Table 2-1. Special Characters

Character	Description
\n	New line
\t	Horizontal tab
\v	Vertical tab
\b	Backspace
\r	Carriage return
\f	Form feed
\\	Backslash character
\'	Single quote
\"	Double quote
\a	Audible or visible alert (control G)
\?	Question mark character '?'

Examples

'a'	*represents the letter a, the value 97*
'\n'	*represents the newline character, the value 10*
'\?'	*represents a question mark, the value 63*
'7'	*represents the character 7, the value 55*
'\0'	*represents the null character, the value 0*
'\101'	*represents the letter A, the value 65*

String Literals

A *string literal* is a sequence of zero or more characters enclosed in double quotation marks.

Syntax

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
       double quote, backslash, or newline
```
escape-sequence

Description

You can type special characters in a character string literal using the escape sequence notation described previously in the section on character constants. The double quote character (`"`) must be represented as an escape sequence if it is to appear inside a string literal. Represent the string '`he said "hi"`' with

```
"he said \"hi\""
```

A character string has static storage duration and type array of `char`.

The actual characters stored in a character string literal are the exact characters specified. In addition, a null character (\0) is automatically added to the end of each character string literal by the compiler. Note that the null character is added only to string literals. Arrays of characters are not terminated with the extra character.

Character string literals that have no characters consist of a single null character.

Note that a string literal containing one character is not the same as a character constant. The string literal `"A"` is stored in two adjacent bytes with the `A` in the first byte and a null character in the second byte; however, the character constant `'A'` is a constant with type int and the value 65 (the ASCII code value for the letter A).

ANSI C allows the usage of wide string literals. A wide string literal is a sequence of zero or more multibyte characters enclosed in double-quotes and prefixed by the letter `L`. A wide string literal has static storage duration and type "array of `wchar_t`." It is initialized with the wide characters corresponding to the given multibyte characters.

Example

```
L"abc##def"
```

Character string literals that are adjacent tokens are concatenated into a single character string literal. A null character is then appended. Similarly, adjacent wide string literal tokens are concatenated into a single wide string literal to which a code with value zero is then appended. It is illegal for a character string literal token to be adjacent to a wide string literal token.

Example

```
char *string = "abc" "def";
```

Operators

An *operator* specifies an operation to be performed on one or more operands.

Syntax

```
operator ::= One selected from the set
    [    ]    (    )    .    ->
    ++   --   &    *    +    -    ~    !    sizeof
    /    %    <<   >>   <    >    <=   >=   !=
    ^    |    &&   ||   ?    :    =    ==   *=
    /=   %=   +=   -=   <<=  >>=  &=   ^=   |=
    ,    #    ##
```

Description

Operator representations may require one, two, or three characters. The compiler matches the longest sequence to find tokens. For example,

```
a+++++b
```

is parsed as if it had been written

```
a++ ++ + b
```

which results in a syntax error. An alternate parse

```
a++ + ++b
```

is not chosen because it does not follow the longest first rule, even though it results in a syntactically correct expression. As a result, white space is often important in writing expressions that use complex operators. The precedence of operators is discussed in more detail in Chapter 5. The obsolete form of the assignment operators (`=*` instead of `*=`) is not supported. If this form is used, the compiler parses it as two tokens (`=` and `*`).

The operators [ ], `?:`, and `( )` (function call operator) occur only in pairs, possibly separated by expressions. You can use some operators as either binary operators or unary operators. Often the meaning of the binary operator is

much different from the meaning of the unary operator. For example, binary multiply and unary indirection:

```
a * b  versus  *p
```

Punctuators

A *punctuator* is a symbol that is necessary for the syntax of the C language, but performs no run-time operation on data and produces no run-time result.

Syntax

punctuator `::=` `One selected from:`

```
[   ]   (   )   {   }   *   ,   :   =   ;   #   ...
```

Description

Some punctuators are the same characters as operators. They are distinguished through the context of their use.

Example

```
#include <stdio.h> /* # marks the processing directive "include"*/
main()              /* ( and ) mark the beginning and end of
                       argument list */

{                   /* { marks the beginning of a block */
   printf("\nHello world\n"); /* ; marks the end of a statement */

}                   /* } marks the end of a block      */
```

Comments

You can include comments to explain code in your program by enclosing the text with `/*` and `*/` characters. If the `/*` character sequence is located within a string literal or a character constant, the compiler processes them as "normal" characters and not as the start of a comment.

You cannot nest comments. To comment blocks of code, enclose the block within the **#if** and **#endif** statements, as shown below:

```
#if 0
:
   code
:
#endif
```

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" later in this chapter. Also, note that it is often valid to define a tag (`struct`, `union`, or `enum`) without actually declaring any objects.

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.

Refer to Chapter 2 for a description of storage duration and linkage.

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 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.

Type 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
struct-or-union specifier
enum-specifier
typedef-name

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

This section describes the *HP specific type qualifier*—`__thread`.

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 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;
```

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`).

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
`volatile int vol_int;`	Declares a volatile int variable.
`const int *ptr_to_const_int;` `int const *ptr_to_const_int;`	Both declare a variable pointer to a constant int.
`int *const const_ptr_to_int`	Declares a constant pointer to a variable int.
`int *volatile vpi, *pi;`	Declares two pointers: vpi is a volatile pointer to an int; pi is a pointer to an int.
`int const *volatile vpci;`	Declares a volatile pointer to a constant int.
`const *pci;`	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;
```

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.

The *HP C 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. Refer to Chapter 10 for the treatment of the sign for bit-fields.

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 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 the section "Enumeration" later in this chapter 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 */
```

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 Workstation and Servers Extension

By default, the HP 9000 workstations and servers HP C compiler 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 `enum`s.

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

Note *enumeration-constant*s 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]

Declarators

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.

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 `int`s. 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 `int`s. 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 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

`int`	*int*
`int *`	*Pointer to int*
`int ()`	*Function returning an int*
`int *()`	*Function returning a pointer to int*
`int (*)()`	*Pointer to function returning an int*

`int [3];`	*Array of 3 int*
`int *[3];`	*Array of 3 pointers to int*
`int (*)[3];`	*Pointer to an array of 3 int*

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];
```

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 initializes. In the case of a structure with automatic storage duration, the initializer may be a single expression returning a 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 initializes, you do not need to specify the size in array declarations. The compiler counts the initializes 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 initializes.

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 initializes 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 initializes, 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"}
};
```

Note that the object being declared (s) is an array of structures without a specified dimension. The compiler counts the initializes to determine the array's dimension. In this case, the presence of two initializes 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();
}
```

Initialization

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 */
}
```

Function Definitions

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.

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 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` 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 may result in 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

This chapter describes forming expressions in C, discusses operator precedence, and provides details about operators used in expressions.

An *expression* in C is a collection of operators and operands that indicates how a computation should be performed. Expressions are represented in infix notation. Each operator has a precedence with respect to other operators. Expressions are building blocks in C. You use the C character set to form tokens. Tokens, combined together, form expressions. Expressions can be used in statements.

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.

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.

Table 5-1 shows the rules of operator precedence in the C language. Table 5-1 lists the highest precedence operators first. 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)
```

Table 5-1. C Operator Precedence

<table>
<tr><th>Operators</th><th>Grouping</th></tr>
<tr><td>() [] -> .</td><td>left to right</td></tr>
<tr><td>+ ! ~ ++ -- - * & sizeof
(See note [1] below.)</td><td>right to left</td></tr>
<tr><td>(type)</td><td>right to left</td></tr>
<tr><td>* / %</td><td>left to right</td></tr>
<tr><td>+ -</td><td>left to right</td></tr>
<tr><td><< >></td><td>left to right</td></tr>
<tr><td>< <= > >=</td><td>left to right</td></tr>
<tr><td>== !=</td><td>left to right</td></tr>
<tr><td>&</td><td>left to right</td></tr>
<tr><td>^</td><td>left to right</td></tr>
<tr><td>|</td><td>left to right</td></tr>
<tr><td>&&</td><td>left to right</td></tr>
<tr><td>||</td><td>left to right</td></tr>
<tr><td>?:</td><td>right to left</td></tr>
<tr><td>= *= /= %= += -= <<=
>>= &= ^= |=</td><td>right to left</td></tr>
<tr><td>,</td><td>left to right</td></tr>
</table>

[1] Note that the `+`, `-`, `*`, and `&` operators listed in this row are unary operators.

Lvalue Expressions

An *lvalue* (pronounced "el-value") is an expression that designates an object. A *modifiable lvalue* is an lvalue that does not have an array or an incomplete type and that does not have a "const"-qualified type.

The term "lvalue" originates from the assignment expression `E1=E2`, where the left operand `E1` must be a modifiable lvalue. It can be thought of as representing an object "locator value." For example, if `E` is the name of an object of static or automatic storage duration, it is an lvalue. Similarly, if `E` denotes a pointer expression, `*E` is an lvalue, designating the object to which `E` points.

Examples

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 */
```

Primary Expressions

The term *primary expression* is used in defining various C expressions.

Syntax

primary-expression ::=
　　identifier
　　constant
　　string-literal
　　(*expression*)

Description

A primary expression is an identifier, a constant, a string literal, or an expression in parentheses that may or may not be an lvalue expression. Primary expressions are the basic components of all expressions.

An identifier can be a primary expression provided that you have declared it properly. A single identifier may or may not be an lvalue expression. A function name is not an lvalue.

A constant is a primary expression and can never be an lvalue.

A string literal is a primary expression. The type of the string literal is "array of characters." If the string literal appears in any context other than as the operand of `sizeof`, the operand of unary `&`, or the initializer for an array of characters, it is converted to a pointer to the first character.

Examples

identifier: `var1`

constant: `99`

string-literal: `"hi there"`

(*expression*) `(a+b)`

Postfix Operators

Postfix operators are unary operators that attach to the end of postfix expressions. A postfix expression is any expression that may be legally followed by a postfix operator.

Syntax

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*

Examples

The following are examples of postfix operators:

The 'element of' operator `([ ])` : `array1[10]`

The postfix increment operator `(++)` : `index++`

The postfix decrement operator `(--)` : `index--`

The argument list of function calls : `func(arg1,arg2,arg3)`

The selection operator `(.)` : `struct_name.member`

The selection operator `(->)` : `p_struct->member`

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.

Array Subscripting

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.

Function Calls

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

Syntax

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

Description

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, as explained in Chapter 4.

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 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 are 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.

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.

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.

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->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)->M`.

Postfix Increment and Decrement Operators

The postfix increment operator `++` adds one to its operand after using its value. The postfix decrement operator `--` subtracts one from its operand after using its value.

Syntax

postfix-expression `++`
postfix-expression `--`

Description

You can only apply postfix increment `++` and postfix decrement `--` operators to an operand that is a modifiable lvalue with scalar type. The result of a postfix increment or a postfix decrement operation is not an lvalue.

The *postfix-expression* is incremented or decremented after its value is used. The expression evaluates to the value of the object *before* the increment or decrement, not the object's new value.

If the value of `X` is 2, after the expression `A=X++` is evaluated, `A` is 2 and `X` is 3.

Avoid using postfix operators on a single operand appearing more than once in an expression. The result of the following example is unpredictable:

```
*p++ = *p++;
```

The C language does not define which expression is evaluated first. The compiler can choose to evaluate the left side of the = operator (saving the destination address) before evaluating the right side. The result depends on the order of the subexpression evaluation.

Pointers are assumed to point into arrays. Incrementing (or decrementing) a pointer causes the pointer to point to the next (or previous) element. This means, for example, that incrementing a pointer to a structure causes the pointer to point to the next structure, *not* the next byte within the structure. (Refer also to "Additive Operators" for information on adding to pointers.)

Unary Operators

You form unary expressions by combining a unary operator with a single operand. All unary operators are of equal precedence and group from right to left.

Syntax

unary-expression `::=`
  postfix-expression
  `++` *unary-expression*
  `--` *unary-expression*
  unary-operator cast-expression
  `sizeof` *unary-expression*
  `sizeof (` *type-name* `)`

unary-operator `::= one selected from`
  `&  *  -  ~  !  +`

Examples

The unary plus operator : `+var`

The unary minus operator : `-var`

The address-of operator : `&var`

The indirect operator : `*ptr`

The logical NOT operator : `!var`

The bitwise NOT operator : `~var`

Prefix Increment and Decrement Operators

The prefix increment or decrement operator increments or decrements its operand before using its value.

Syntax

`++` *unary-expression*
`--` *unary-expression*

Description

The operand for the prefix increment `++` or the prefix decrement `--` operator must be a modifiable lvalue with scalar type. The result is not an lvalue.

The operand of the prefix increment operator is incremented by 1. The resulting value is the result of the *unary-expression*.

The prefix decrement operator behaves the same way as the prefix increment operator except that a value of one is subtracted from the operand.

For any expression E, the unary expressions `++E` and `(E+=1)` yield the same result. If the value of `X` is 2, after the expression `A=++X` is evaluated, `A` is 3 and `X` is 3.

Pointers are assumed to point into arrays. Incrementing (or decrementing) a pointer causes the pointer to point to the next (or previous) element. This means, for example, that incrementing a pointer to a structure causes the pointer to point to the next structure, *not* the next byte within the structure. (Refer also to "Additive Operators" for information on adding to pointers.)

Address and Indirection Operators

The address (&) and indirection (*) operators are used to locate the address of an operand and indirectly access the contents of the address.

Syntax

& *cast-expression*
* *cast-expression*

Description

The operand of the unary indirection operator (*) must be a pointer to type T. The resulting type is T. If type T is not a function type, the *unary-expression* is an lvalue.

The contents of pointers are memory addresses. No range checking is done on indirection operations. Specifically, storing values indirectly through a pointer that was not correctly initialized can cause bounds errors or destruction of valid data.

The operand of the unary address-of operator (&) must be a function designator or an lvalue. This precludes taking the address of constants (for example, `&3`), because `3` is not an lvalue. If the type of the operand is T, the result of the address of operator is a pointer to type T. The `&` operator may not be applied to bit fields or objects with the `register` storage class.

It is always true that if `E` is an lvalue, then `*&E` is an lvalue expression equal to `E`.

Unary Arithmetic Operators

A unary arithmetic operator combined with a single operand forms a unary expression used to negate a variable, or determine the ones complement or logical complement of the variable.

Syntax

`+` *cast-expression*
`-` *cast-expression*
`~` *cast-expression*
`!` *cast-expression*

Description

The unary plus operator operates on a single arithmetic operand, as is the case of the unary minus operator. The result of the unary plus operator is defined to be the value of its operand. For example, just as `-2` is an expression with the value negative 2, `+2` is an expression with the value positive 2.

In spite of its definition, the unary plus operator is not purely a no-op. According to the ANSI standard, an unary plus operation is an expression that follows the integral promotion rule. For example, if `i` is defined as a short int, then `sizeof (i )` is 2. However, `sizeof (+i)` is 4 because the unary plus operator promotes `i` to an int. The result of the unary `-` operator is the negative value of its operand. The operand can be any arithmetic type. The integral promotion is performed on the operand before it is used. The result has the promoted type and is not an lvalue.

The result of the unary `~` operator is a one's (bitwise) complement of its operand. The operand can be of any integral type. The integral promotion is performed on the operand before it is used. The result has the promoted type and is not an lvalue.

The result of the unary `!` operator is the logical complement of its operand. The operand can be of any scalar type. The result has type int and is not an lvalue. If the operand had a zero value, the result is 1. If the operand had a nonzero value, the result is 0.

The sizeof Operator

The `sizeof` operator is used to determine the size (in bytes) of a data object or a type.

Syntax

`sizeof` *unary-expression*
`sizeof` (*type-name*)

Description

The result of the `sizeof` operator is an `unsigned int` constant expression equal to the size of its operand in bytes. You can use the `sizeof` operator in two different ways. First, you can apply the `sizeof` operator to an expression. The result is the number of bytes required to store the data object resulting from the expression. Second, it may be followed by a type name inside parentheses. The result then is the number of bytes required to store the specified type.

In either usage, the `sizeof` operator is a compile-time operator that you can use in place of an integer constant.

The usual conversion of arrays of T to pointers to T is inhibited by the `sizeof` operator. The `sizeof` operator returns the number of bytes in an array rather than the number of bytes in a pointer.

When you apply the `sizeof` operator to an expression, the expression is not compiled into executable code. This means that side effects resulting from expression evaluation do not take place.

Cast Operators

The cast operator is used to convert an expression of one type to another type.

Syntax

cast-expression ::=
    unary-expression
    (*type-name*) *cast-expression*

Description

An expression preceded by a parenthesized type name causes the expression to be converted to the named type. This operation is called a *cast*. The cast does not alter the type of the expression, only the type of the value. Unless the type name specifies `void` type, the type name must specify a scalar type, and the operand must have scalar type.

The result of a cast operation is not an lvalue.

Conversions involving pointers (other than assignment to or from a "pointer to `void`" or assignment of a null pointer constant to a pointer) require casts.

A pointer can be cast to an integral type and back again provided the integral type is at least as wide as an int.

A pointer to any object can safely be converted to a pointer to `char` or a pointer to `void`, and back again. If converted to a pointer to `char`, it will point to the first (lowest address) byte of the original object. For example, a pointer to an integer converted to a character pointer points to the most significant byte of the integer.

A pointer to a function of one type can safely be converted to a pointer to a function of another type, and back again.

Multiplicative Operators

The *multiplicative operators* perform multiplication (`*`), division (`/`), or remainder (`%`).

Syntax

multiplicative-expression `::=`
  cast-expression
  multiplicative-expression `*` *cast-expression*
  multiplicative-expression `/` *cast-expression*
  multiplicative-expression `%` *cast-expression*

Description

Each of the operands in a multiplicative expression must have arithmetic type. Further, the operands for the `%` operator must have integral type.

The usual arithmetic conversions are performed on the operands to select a resulting type. The result is not an lvalue.

The result of the multiplication operator `*` is the arithmetic product of the operands.

The result of the division operator `/` is the quotient of the operands.

The result of the mod operator `%` is the remainder when the left argument is divided by the right argument. By definition, `a%b==a- ((a/b)*b)`. The second operand (`/` or `%`) must not
be 0.

The following table describes the result of `a/b` for positive and negative integer operands, when the result is inexact.

	b positive	b negative
a positive	Largest integer less than the true quotient.	Smallest integer greater than the true quotient.
a negative	Smallest integer greater than the true quotient.	Largest integer less than the true quotient.

For example, `-5/2 == -2`. The true quotient is -2.5; the smallest integer greater than -2.5 is -2.

The following table describes the sign of the result of `a%b` for positive and negative operands, when the result is not zero.

	b positive	b negative
a positive	+	+
a negative	–	–

For example:

```
-5 % 2 == -1
```

Examples

```
var1 * var2

var1 / var2

var1 % var2
```

Additive Operators

The *additive operators* perform addition (+) and subtraction (-).

Syntax

additive-expression ::=
  multiplicative-expression
  additive-expression + *multiplicative-expression*
  additive-expression - *multiplicative-expression*

Description

The result of the binary addition operator + is the sum of two operands. Both operands must be arithmetic, or one operand can be a pointer to an object type and the other an integral type. The usual arithmetic conversions are performed on the operand if both have arithmetic type. The result is not an lvalue.

If one operand is a pointer and the other operand is an integral type, the integral operand is scaled by the size of the object pointed to by the pointer. As a result, the pointer is incremented by an integral number of objects (not just an integral number of storage units). For example, if `p` is a pointer to an object of type T, when the value 1 is added to `p`, the value of 1 is scaled appropriately. Pointer `p` will point to the next object of type T. If any integral value `i` is added to `p`, `i` is also scaled so that `p` will point to an object that is `i` objects away since it is assumed that `p` actually points into an array of objects of type T. Use caution with this feature. Consider the case where `p` points to a structure that is ten bytes long. Adding a constant 1 to `p` does not cause `p` to point to the second byte of the structure. Rather it causes `p` to point to the next structure. The value of one is scaled so a value of ten (the length in bytes of the structure) is used.

The result of the binary subtraction operator - is the difference of the two operands. Both operands must be arithmetic; the left operand can be a pointer and the right can be an integral type; or both must be pointers to the same type. The usual arithmetic conversions are performed on the operands if both have arithmetic type. The result is not an lvalue.

If one operand is a pointer and the other operand is an integral type, the integral operand is scaled by the size of the object pointed to by the left operand. As a result, the pointer is decremented by an integral number of objects (not just an integral number of storage units). See the previous discussion on the addition operator + for more details.

If both operands are pointers to the same type of object, the difference between the pointers is divided by the size of the object they point to. The result, then, is the integral number of objects that lie between the two pointers. Given two pointers `p` and `q` to the same type, the difference `p-q` is an integer `k` such that adding `k` to `q` yields `p`.

Examples

```
var1+var2

var1-var2
```

Bitwise Shift Operators

The *bitwise shift operators* shift the left operand left (<<) or right (>>) by the number of bit positions specified by the right operand.

Syntax

shift-expression ::=
    additive-expression
    shift-expression << *additive-expression*
    shift-expression >> *additive-expression*

Description

Both operands must be of integral type. The integral promotions are performed on both operands. The type of the result is the type of the promoted left operand.

The left shift operator << shifts the first operand to the left and zero fills the result on the right. The right shift operator >> shifts the first operand to the right. If the type of the left operand is an unsigned type, the >> operator zero fills the result on the left. If the type of the left operand is a signed type, copies of the sign bit are shifted into the left bits of the result (sometimes called *sign extend*).

Example

```
var1>>var2
```

Relational Operators

The *relational operators* compare two operands to determine if one operand is less than, greater than, less than or equal to, or greater than or equal to the other.

Syntax

relational-expression `::=`
    shift-expression
    relational-expression `<` *shift-expression*
    relational-expression `>` *shift-expression*
    relational-expression `<=` *shift-expression*
    relational-expression `>=` *shift-expression*

Description

The usual arithmetic conversions are performed on the operands if both have arithmetic type. Both operands must be arithmetic or both operands must be pointers to the same type. In general, pointer comparisons are valid only between pointers that point within the same aggregate or union.

Each of the operators `<` (less than), `>` (greater than), `<=` (less than or equal) and >= (greater than or equal) yield 1 if the specified relation is true; otherwise, they yield 0. The resulting type is `int` and is not an lvalue.

When two pointers are compared, the result depends on the relative locations in the data space of the objects pointed to. Pointers are compared as if they were unsigned integers.

Because you can use the result of a relational expression in an expression, it is possible to write syntactically correct statements that appear valid but which are not what you intended to do. An example is `a<b<c`. This is not a representation of "`a` is less than `b` and `b` is less than `c`." The compiler interprets the expression as `(a<b)<c`. This causes the compiler to check whether `a` is less than `b` and then compares the result (an integer 1 or 0) with `c`.

Examples

```
var1 < var2

var1 > var2

var1 <= var2

var1 >= var2
```

Equality Operators

The *equality operators* equal-to (==) and not-equal-to (!=) compare two operands.

Syntax

equality-expression ::=
  relational-expression
  equality-expression == *relational-expression*
  equality-expression != *relational-expression*

Description

The usual arithmetic conversions are performed on the operands if both have arithmetic type. Both operands must be arithmetic, or both operands must be pointers to the same type, or one operand can be a pointer and the other a null pointer constant or a pointer to void.

Both of the operators == (equal) and != (not equal) yield 1 if the specified relation is true; otherwise they will yield 0. The result is of type int and is not an lvalue.

The == and != operators are analogous to the relational operators except for their lower precedence. This means that the expression a<b==c<d is true if and only if a<b and c<d have the same truth value.

Use caution with the == operator. It resembles the assignment operator (=) and is often pronounced the same when programs are read. Further, you can use the == operator in expressions syntactically the same as you would the = operator. For example, the statements

```
if(a==b) return 0;

if(a=b) return 0;
```

look very much alike, but are very different. The first statement says "if a is equal to b, return a value of zero." The second statement says "store b into a and if the value stored is nonzero, return a value of zero."

Examples

```
var1==var2
```

```
var1!=var2
```

Bitwise AND Operator

The *bitwise AND operator* (**&**) performs a bitwise AND operation on its operands. This operation is useful for bit manipulation.

Syntax

AND-expression `::=`
    equality-expression
    AND-expression `&` *equality-expression*

Description

The result of the binary **&** operator is the bitwise AND function of the two operands. Both operands must be integral types. The usual arithmetic conversions are performed on the operands. The type of the result is the converted type of the operands. The result is not an lvalue.

For each of the corresponding bits in the left operand, the right operand, and the result, the following table indicates the result of a bitwise AND operation.

Bit in Left Operand	Bit in Right Operand	Bit in Result
0	0	0
0	1	0
1	0	0
1	1	1

Example

```
var1 & var2
```

Bitwise Exclusive OR Operator

The *bitwise exclusive OR operator* (^) performs the bitwise exclusive OR function on its operands.

Syntax

exclusive-OR-expression ::=
    AND-expression
    exclusive-OR-expression ^ *AND-expression*

Description

The result of the binary operator ^ is the bitwise exclusive OR function of the two operands. Both operands must be integral types. The usual arithmetic conversions are performed on the operands. The type of the result is the converted type of the operands. The result is not an lvalue.

For each of the corresponding bits in the left operand, the right operand, and the result, the following table indicates the result of an exclusive OR operation.

Bit in Left Operand	Bit in Right Operand	Bit in Result
0	0	0
0	1	1
1	0	1
1	1	0

You can use the exclusive OR operation for complementing bits. If a mask integer is exclusive OR'd with another integer, each bit position in the mask having a value of one will cause the corresponding position in the other operand to be complemented.

Example

```
var1 ^ var2
```

Bitwise Inclusive OR Operator

The *bitwise inclusive OR operator* (|) performs the bitwise inclusive OR function on its operands.

Syntax

inclusive-OR-expression ::=
  exclusive-OR-expression
  inclusive-OR-expression | *exclusive-OR-expression*

Description

The result of the binary operator | is the bitwise OR function of the two operands. Both operands must be integral types. The usual arithmetic conversions are performed on the operands. The type of the result is the converted type of the operands. The result is not an lvalue.

For each of the corresponding bits in the left operand, the right operand, and the result, the following table indicates the result of a bitwise OR operation.

Bit in Left Operand	Bit in Right Operand	Bit in Result
0	0	0
0	1	1
1	0	1
1	1	1

Example

```
var1 | var2
```

Logical AND Operator

The *logical AND operator* (`&&`) performs the logical AND function on its operands.

Syntax

logical-AND-expression `::=`
    inclusive-OR-expression
    logical-AND-expression `&&` *inclusive-OR-expression*

Description

Each of the operands must have scalar type. The type of the left operand need not be related to the type of the right operand. The result has type `int` and has a value of 1 if both of its operands compare unequal to 0, and 0 otherwise. The result is not an lvalue.

The logical AND operator guarantees left-to-right evaluation. If the first operand compares equal to zero, the second operand is not evaluated.

This feature is useful for pointer operations involving pointers that can be NULL. For example, the following statement:

```
if(p!=NULL && *p=='A') *p='B';
```

The first operand tests to see if pointer `p` is NULL. If `p` is NULL, an indirect reference could cause a memory access violation. If `p` is non-NULL, the second operand is safe to evaluate. The second expression checks to see if `p` points to the character `'A'`. If the second expression is true, the `&&` expression is true and the character that `p` points to is changed to `'B'`. Had the pointer been NULL, the `if` statement would have failed and the pointer would not be used indirectly to test for the 'A' character.

Example

```
var1 && var2
```

Logical OR Operator

The *logical OR operator* (`||`) performs the logical OR function on its operands.

Syntax

logical-OR-expression `::=`
  logical-AND-expression
  logical-OR-expression `||` *logical-AND-expression*

Description

Each of the operands must be of scalar type. The type of the left operand need not be related to the type of the right operand. The result has type `int` and has a value of 1 if either of its operands compare unequal to 0, and 0 otherwise. The result is not an lvalue.

The logical OR operator guarantees left-to-right evaluation. If the first operand compares unequal to 0, the second operand is not evaluated.

Example

```
var1 || var2
```

Conditional Operator

The *conditional operator* (**?:**) performs an if-then-else using three expressions.

Syntax

conditional-expression ::=
    logical-OR-expression
    logical-OR-expression ? *expression* : *conditional-expression*

Description

A conditional expression consists of three expressions. The first and the second expressions are separated with a **?** character; the second and third expressions are separated with a **:** character.

The first expression is evaluated. If the result is nonzero, the second expression is evaluated and the result of the conditional expression is the value of the second expression. If the first expression's result is zero, the third expression is evaluated and the result of the conditional expression is the value of third expression.

The first expression can have any scalar type. The second and third expressions can be any of the following combinations:

1. **Both arithmetic.**

   The usual arithmetic conversions are performed on the second and third expressions. The resulting type of the conditional expression is the result of the conversion.

2. **Both are pointers to type T.**

   Arrays are converted to pointers, if necessary. The result is a pointer to type T.

3. **Identical type object.**

   The types can match and be structure, union, or `void`. The result is that specific type.

4. **Pointer and Null pointer constant or a pointer to void**

   One expression may be a pointer (or array that is converted to a pointer) and the other a null pointer constant or a pointer to void. The result is the same type as the type of the pointer operand.

In all cases, the result is not an lvalue.

Note that *either* the second *or* the third expression is evaluated, but not both. Although not required for readability, it is considered good programming style to enclose the first expression in parentheses. For example:

```
min = (val1<val2) ? val1:val2;
```

Example

This expression returns `x` if `a` is 0, or return `y` if `a` is not 0.

```
a == 0 ? x : y
```

The following statement prints `"I have 1 dog."` if `num` is equal to 1, or `"I have 3 dogs."`, if `num` is 3.

```
printf ("I have %d dog%s.\n",num, (num>1) ? "s" : "");
```

Assignment Operators

Assignment operators assign the value of the right operand to the object designated by the left operand.

Syntax

assignment-expression `::=`
  conditional-expression
  unary-expression assignment-operator assignment-expression

assignment-operator `::= one selected from the set`
  `= *= /= %= += -= <<= >>= &=  ^= |=`

Description

Each assignment operator must have a modifiable lvalue as its left operand. An assignment operator stores a value into the left operand. The C language does not define the order of evaluation of the left and right operands. For this reason, you should avoid operations with side effects (such as `++` or `--`) if their operands appear on both the left and right side of the assignment. For example, you should not write an expression like the following because the results depend on which operand is evaluated first.

```
*p++ = *p--
```

Simple Assignment

In simple assignment, the value of the right operand replaces the value of the object specified by the left operand. If the source and destination objects overlap storage areas, the results of the assignment are undefined.

The left and right operands can be any of the following combinations:

1. **Both arithmetic**

   If both of the operands are arithmetic types, the type of the right operand is converted to the type of the left operand. The converted value is then stored in the location specified by the left operand.

2. **Both structure/union**

   If both operands are structures or unions of the same type, the right structure/union is copied into the left structure/union. A union is considered to be the size of the largest member of the union, and it is this number of bytes that is moved.

3. **Left operand is a pointer to type T**

   In this case, the right operand can also be a pointer to type `T`. The right operand is then copied to the left operand.

   The right operand can also be a null pointer constant or a pointer to void.

   A special case of pointer assignment involves the assignment of a pointer to `void` to another pointer. No cast is necessary to convert a "pointer to `void`" to any other type of pointer.

An assignment is not only an operation, it is also an expression. Each operand must have an arithmetic type consistent with those allowed by the binary operator that is used to make up the assignment operator. You can use the `+=` and `-=` operators with a left operand that is a pointer type.

Compound Assignment

Given the general assignment operator *op*=, if used in the expression

```
A op= B
```

the result is equal to the following assignment

```
A = A op (B)
```

except that the expression represented by `A` is evaluated only once.

Therefore,

```
A[f()] += B
```

is very different from

```
A[f()] = A[f()] + B
```

because the latter statement causes the function `f()` to be invoked twice.

Assignment Operators

Assignment operators are useful to reference complex subscript operators. For example:

```
a[j+2/i] += 3.5
```

In this case, the subscript expression is evaluated only once.

Examples

`a += 5`	*Add 5 to a.*
`a *= 2`	*Multiply a by 2.*
`a = b`	*Assign b to a.*
`a <<= 1`	*Left shift a by 1 bit.*

Comma Operator

The *comma operator* is a binary operator whose operands are expressions. The expression operands are evaluated from left to right.

Syntax

expression `::=`
  assignment-expression
  expression `,` *assignment-expression*

Description

The comma operator is a "no-operation" operator. Its left operand is evaluated for its side effects only. Its value is discarded. The right operand is then evaluated and the result of the expression is the value of the right operand.

Because the comma is a punctuator in lists of function arguments, you need to use care within argument lists to ensure that the comma is treated as a comma operator and not as an argument separator.

```
f(a, (b=7, b), c);
```

This example passes three arguments to `f()`. The first is the value of `a`, the second is the value of `b` which is set equal to 7 before the function call, and the third is the value of `c`. The comma separating the assignment expression and the argument `b` is enclosed in parentheses. It is therefore interpreted as a comma operator and not as an argument separator.

Examples

```
func(a, (b=0, b), c) /*Set b to 0 before passing it to func. */

index++, a = index  /*Increment index and then assign it to a.*/

i=0, j=0, k=0       /*Initialize i,j,k to 0 */
```

Constant Expressions

Constant expressions are expressions that can be evaluated during translation rather than run-time.

Syntax

constant-expression ::=
    conditional-expression

Description

A constant expression must evaluate to an arithmetic constant expression, a null pointer constant, an address constant, or an address constant plus or minus an integral constant expression.

An integral constant expression must involve only integer constants, enumeration constants, character constants, `sizeof` expressions, and casts to integral types. You cannot include the array subscripting (`[ ]`), member access (`.` and `->`), and address of (`&`) operators in integral constant expressions. An integral constant expression with the value 0, or such an expression cast to type `void *`, is called a *null pointer constant*. An *address constant* is a pointer to an object of static storage duration or to a function designator.

Further, you cannot use a function call, an increment or a decrement operator, or indirection or assignment operations in a constant expression.

Constant expressions are usually used for "allocation" type operations. An example of this is array allocation. The size of an array is given as a constant expression.

Examples

```
2 * 2
3 + 3
(-2.5) + 99.8 * 4.5
```

6

Statements

This chapter describes the statements in the C programming language. The statements are grouped as follows:

- Labeled Statements.
- Compound Statement or Block.
- Expressions and Null Statement.
- Selection Statements.
- Iteration Statements.
- Jump Statements.

Statements are the executable parts of a C function. The computer executes them in the sequence in which they are presented in the program, except where control flow is altered as specified in this chapter.

Syntax

statement ::=
labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

Example

```
labl: x=y;
{ int x; x=y; }
x=y;
if (x<y) x=y;
for (i=1; i<10; i++) a[i]=i;
goto labl;
```

Labeled Statements

Labeled statements are those preceded by a label.

Syntax

labeled-statement ::=
  identifier : *statement*
  `case` *constant-expression* : *statement*
  `default:` *statement*

Description

You can prefix any statement using a label so at some point you can reference it using `goto` statements. This includes statements already having labels. In other words, any statement can have one or more labels affixed to it.

The `case` and `default` labels can only be used inside a `switch` statement. These are discussed in further detail in the section on the `switch` statement appearing later in this chapter.

Examples

```
if (fatal_error)
   goto get_out;
   :
get_out: return(FATAL_CONDITION);
```

Compound Statement or Block

Compound or block statements allow you to group other statements together in a block of code.

Syntax

compound-statement ::=
  {[*declaration-list*] [*statement-list*]}

declaration-list ::=
  declaration
  declaration-list declaration

statement-list ::=
  statement
  statement-list statement

Description

You can group together a set of declarations and statements and use them as if they were a single statement. This grouping is called a *compound statement* or a *block*.

Variables and constants declared in the block are local to the block and any subordinate blocks declared therein 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 a subordinate block.

Example

```
if (x != y)
{
      int temp;
   temp = x;
   x = y;
   y = temp;
}
```

Expression and Null Statements

An expression in C can also be a statement. A *null statement* is an expression statement with no expression and it is used to provide a null operation.

Syntax

expression-statement ::=
    [*expression*];

Description

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. For 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++]);
```

No additional statement is necessary to specify the required function, however, the syntax of the `for` statement requires a statement following the closing) of the `for`. To meet this syntax requirement, you can use a null statement.

Example

expression;: `x=y;`

Selection Statements

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.

Syntax

selection-statement `::=`
  `if (`*expression*`)` *statement*
  `if (`*expression*`)` *statement* `else` *statement*
  `switch (`*expression*`)` *statement*

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;
}
```

The if Statement

The `if` statement executes a statement depending on the evaluation of an expression.

Syntax

`if (`*expression* `)` *statement*
`if (`*expression* `)` *statement* `else` *statement*

Description

The `if` statement is for testing values and making decisions. An `if` statement can optionally include an `else` clause. For example:

```
if (j<1)
    func(j);
else
   {
    j=x++;
    func(j);
   }
```

The first statement is executed only if the evaluated expression is true (in other words, evaluates to a nonzero value). The expression may be of any scalar type. Note that expressions involving relational expressions actually produce a result and may therefore be used in an `if` statement.

If you include the `else` clause, the statement after the `else` is executed only if the evaluated expression is false (in other words, evaluates to a zero value). Under no circumstances are both statements in an `if-else` statement executed (unless you include a `goto` statement from one substatement to the other).

If the first substatement is entered as the result of a `goto` to a label, the second substatement (if provided) is not executed.

The "dangling else" ambiguity associated with `if` statements of this form is resolved by associating the `else` with the last lexically preceding `else`-less `if` that is in the same block, but not in an enclosed block.

The `else-if` construction is useful to include more than one alternative to the `if` statement. The following is an example of a three-way branch using the `else-if` chain:

```
if(a==b)
   k = 1;
else if(a==c)
   k = 2;
else if(a==d)
   k = 4;
```

Regardless of the relationships between the variables `a, b` and `c`, only one statement assigning a value to `k` is executed. You should use the `else-if` chain in place of the `switch` statement when the controlling expressions are not constant expressions. However, nesting too many `else-if` statements can make a program cumbersome.

The tests are each executed in order until successful or until the end of the selection statement is reached. In the previous example, if `a` is equal to `d`, all three comparisons would be executed. On the other hand, if `a` is equal to `c`, only the first two comparisons are executed. Therefore, conditions that are most likely to be true should be tested first in an `else-if` chain. The `switch` statement, however, may execute only one comparison (depending on efficiency tradeoffs). Use the `switch` statement where possible to make a program more readable and efficient (see "The switch Statement").

The switch Statement

The `switch` statement executes one or more of a series of cases based on the value of an expression. It offers multiway branching.

Syntax

```
switch (expression)
    statement
```

Description

The *expression* after the word `switch` is the *controlling expression*. The controlling expression must have integral type. The statement following the controlling expression is typically a compound statement. The compound statement is called the *switch body*.

The statements in the switch body may be labeled with `case` labels. The `case` label is followed by an integral constant expression and a colon. No two `case` constant expressions in the same switch statement may have the same value.

When the `switch` statement executes, integral promotions are performed on the controlling expression; the result is compared with the constant expressions after the case labels in the `switch body`. If one of the constant expressions matches the value of the controlling expression, control passes to the statement following that case expression.

If no expression matches the value of the control expression and a statement in the switch body is labeled with the `default` label, control passes to that statement. Only one statement of the switch body may be labeled the `default`. By convention, the `default` label is included last after the `case` labels, although this is not required by the C programming language.

If there is no `default`, control passes to the statement immediately following the `switch` body and the `switch` effectively becomes a no- operation statement.

The `switch` statement operates like a multiway `else-if` chain except the values in the `case` statements must be constant expressions, and, most importantly, once a statement is selected from within the `switch body`, control

is passed from statement to statement as in a normal C program. Control may "fall" through to following case statements. Using a `break` statement is the most common way to leave a switch body. If a `break` statement is encountered, control passes to the statement immediately following the `switch` body.

Example

The following example shows a `switch` statement that includes several case labels. The program selects the case whose constant matches `getchar`.

```
  switch (getchar ( ) )
{
      case 'r':
      case 'R':
         moveright ( );
         break;
      case 'l':
      case 'L':
         moveleft ( );
         break;
      case 'B':
         moveback ( );
         break;
      case 'A':
      default:
         moveahead ( );
         break;
}
```

Iteration Statements

You use iteration statements to force a program to execute a statement repeatedly. The executed statement is called the *loop body*. Loops execute until the value of a controlling expression is 0. The controlling expression may be of any scalar type.

C has several iteration 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.

Syntax

iteration-statement `::=`
  `while (`*expression*`)` *statement*
  `do` *statement* `while (`*expression*`);`
  `for (`*[expression1]* `;` *[expression2]*`;` *[expression3]*`)` *statement*

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;
}
```

The while Statement

The `while` statement evaluates an expression and executes the loop body until the expression evaluates to false.

Syntax

```
while (expression)
    statement
```

Description

The controlling expression is evaluated at run time. If the controlling expression has a nonzero value, the loop body is executed. Then control passes back to the evaluation of the controlling expression. If the controlling expression evaluates to 0, control passes to the statement following the loop body. The test for 0 is performed before the loop body is executed. The loop body of a `while` statement with a controlling constant expression that evaluates to 0 never executes.

Example

For example:

```
i = 0;
while (i < 3) {
    func (i);
    i++;
}
```

The example shown above calls the function `func` three times, with the argument values of 0, 1, and 2.

The do Statement

The `do` statement executes the loop body one or more times until the expression in the `while` clause evaluates to 0.

Syntax

`do` *statement* `while (`*expression*`)`

Description

The loop body is executed. The controlling expression is evaluated. If the value of the expression is nonzero, control passes to the first statement of the loop body. Note that the test for a zero value is performed after execution of the loop body. The loop body executes one time regardless of the value of the controlling expression.

Example

For example:

```
i = 0;
do {
     func (i);
     i++;
} while (i<3);
```

This example calls the function `func` three times with the argument values of 0, 1, and 2.

The for Statement

The `for` statement evaluates three expressions and executes the loop body until the second expression evaluates to false.

Syntax

`for (`*[expression1]* ; *[expression2]*; *[expression3]*`)` *statement*

Description

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.

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 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 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.

The `for` loop continues to execute until *expression2* evaluates to 0, or until a jump 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:

The for Statement

```
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.

Example

For example:

```
for (i=0; i<3; i++) {
    func(i);
}
```

This example calls the function `func` three times, with argument values of 0, 1, and 2.

Jump Statements

Jump statements cause the unconditional transfer of control to another place in the executing program.

Syntax

jump-statement ::=
  goto *identifier*;
  continue;
  break;
  return *[expression]*;

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  */
   }
}
```

The goto Statement

The `goto` statement transfers control to a labeled statement that is within the scope of the current function.

Syntax

`goto` *identifier*;

Description

The `goto` statement causes an unconditional branch to the named label in the current function. Because you can use `goto` statements to jump to any statement that can be labeled, the potential for their abuse is great. For example, you can branch into loop bodies and enter blocks at points other than the head of the block. This can cause problems if you attempt to access variables initialized at the beginning of the block. Generally, you should avoid using `goto` statements because they disturb the structure of the program, making it difficult to understand. A common use of `goto` statements in C is to exit from several levels of nested blocks when detecting an error.

The continue Statement

The `continue` statement is used to transfer control during the execution of an iteration statement.

Syntax

```
continue;
```

Description

The `continue` statement unconditionally transfers control to the loop-continuation portion of the most tightly enclosing iteration statement. You cannot use the `continue` statement without an enclosing `for`, `while`, or `do` statement.

In a `while` statement, a `continue` causes a branch to the code that tests the controlling expression.

In a `do` statement, a `continue` statement causes a branch to the code that tests the controlling expression.

In a `for` statement, a `continue` causes a branch to the code that evaluates the increment expression.

Example

For example:

```
for (i=0; i<=6; i++)
     if(i==3) continue;
     else printf("%d\n",i);
```

This example prints:

```
0
1
2
4
5
6
```

The break Statement

The `break` statement terminates the enclosing switch or iteration statement.

Syntax

```
break;
```

Description

A `break` statement terminates the execution of the most tightly enclosing `switch` or iteration statement. Control is passed to the statement following the `switch` or iteration statement. You cannot use a `break` statement unless it is enclosed in a `switch` or iteration statement. Further, a `break` will only break out of one level of `switch` or iteration 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

For example:

```
for (i=0; i<=6; i++)
     if(i==3) break;
     else printf ("%d\n",i);
```

This example prints:

```
     0
     1
     2
```

The return Statement

The `return` statement causes a return from a function.

Syntax

```
return [expression];
```

Description

When a `return` statement is executed, the current function is terminated and control passes back to the calling function. In addition, all memory previously allocated to automatic variables is considered unused and may be allocated for other purposes.

If an expression follows the `return` statement, the value of the expression is implicitly cast to match the type of the function in which the `return` statement appears. If the type of the function is `void,` no expression may follow the `return` statement.

A given function may have as many `return` statements as necessary. Each may have an expression or not, as required. Note that the C language does not require that `return` statements have expressions even if the function type is not `void`. If a calling program expects a value and a function does not return one (that is, a `return` statement has no expression), the value returned is undefined.

Reaching the final } character of a function without encountering a `return` is equivalent to executing a `return` statement with no expression.

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 use the preprocessing directives to make a number of textual changes in the source before it is syntactically and semantically analyzed and translated. Since preprocessing occurs conceptually before the compilation process, there is generally no relationship between the syntax of a translation unit and preprocessing directives. There are some restrictions on where `#pragma` directives may appear within a translation unit. Refer to Chapter 9 for details.

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

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

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.

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" below), 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
`__DATE__`	Produces the date of compilation in the form `Mmm dd yyyy`.
`__FILE__`	Produces the name of the file being compiled.
`__LINE__`	Produces the current source line number.
`__STDC__`	Produces the decimal constant 1, indicating that the implementation is standard-conforming.
`__TIME__`	Produces the time of compilation in the form `hh:mm:ss`.

Note `__DATE__`, `__TIME__`, and `__STDC__` are only defined in ANSI mode.

Conditional Compilation

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
```

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
#ifndef 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 */

#if defined(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 */
                        /* defined to be 16 (bits).            */

  #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
```

Note The `#elif` directive is only supported in ANSI mode.

Line Control

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

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 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

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
#endif                            the diagnostic message "HP_C
                                  not defined!"              */

#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

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

Any `?` that does not begin one of the trigraphs listed above is not changed.

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.

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 workstation and server programs on the HP-UX operating system. 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 `LPATH` 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 `LPATH` or on creating and linking with shared libraries.) Object modules are combined into an executable program file.

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 "Compiler Options" in this chapter 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 under Table 9-2 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 _HPUX_SOURCE name space macro. It is equivalent to -Aa +e -D_HPUX_SOURCE.
-c	Compiles only, does not link.
-C	Prevents the preprocessor from stripping comments.
-D*name*	Defines the preprocessor variable *name* with a value of "1".
-D*name=def*	Defines the preprocessor variable *name* with a value of *def*.
-E	Performs preprocessing only with output to stdout.
-g	Inserts information for the symbolic debugger in the object file.
-G	Inserts information required by the gprof profiler in the object file.
-I*dir*	Inserts *dir* in the include file search path.
-l*x*	Links with the /lib/lib*x*.a and /usr/lib/lib*x*.a libraries.
-L*dir*	Links the libraries in *dir* before the libraries in the default search path.
-n	Generates shareable code.
-N	Generates unshareable code.
-o *outfile*	Places object modules in *outfile* file.
-O	Optimizes at level 2.
-p	Inserts information required by the prof profiler in the object file.
-P	Performs preprocessing only with output to the corresponding *.i* file.
-q	Marks the executable as demand loadable.

Table 9-1. HP C Compiler Options at a Glance (continued)

Option	Description
-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 *x,name*	Substitutes or inserts subprocess *x* with *name*.
-U*name*	Undefines *name* in the preprocessor.
-v	Enables verbose mode.
-V	Causes subprocesses to print version information to **stderr**.
-w	Suppresses warning messages.
-Wd,-a or +a	Omits HP provided prefix files required by the linker.
-W*x*, *arg1* *[,arg2,..,argn]*	Passes the arguments *arg1* through *argn* to the subprocess *x*.
-y	Generates information used by the *HP SoftBench* 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.
+DA*model*	Generates object code for a specific version of the PA-RISC architecture.
+df*name*	Specifies the profile database to use with profile-based optimization.
+DS*model*	Performs instruction scheduling for a specific implementation of PA-RISC.
+e	Enables the following HP value added features while compiling in ANSI C mode: **sized enum**, **long long**, long pointers, compiler supplied defaults for missing arguments to intrinsic calls, and **$** in identifier HP C extensions.

Table 9-1. HP C Compiler Options at a Glance (continued)

Option	Description
`+ESfic`	Replaces millicode calls with inline fast indirect calls.
`+ESlit`	Places string literals and constants into read-only data storage.
`+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`*flags*	Controls floating-point traps.
`+help`	Invokes the initial menu window of the online programmer's guide.
`+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`.
`+o`	Prints hexadecimal code offsets in the source code listing.
`+O`*opt*	Invokes optimization level `opt` where `opt` is 0 to 4. See the *HP C Programmer's Guide* for additional optimization options.
`+P`	Performs profile-based optimization.

Table 9-1. HP C Compiler Options at a Glance (continued)

Option	Description
`+pgm`*name*	Specifies the execution profile data set to be used by the optimizer.
`+r`	Inhibits the automatic promotion of `float` to `double`.
`+u`*bytes*	Controls pointer alignment where *bytes* is 1, 2, or 4.
`+w`*n*	Specifies the level of the warning messages where *n* is 1 - 3.
`+z`	Generates shared library object code.
`+Z`	Generates shared library object code with a large data linkage table.

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
`-A`*level*	where *level* can be `a`, `c` or `e`.

`a`	Requests a compilation in ANSI C mode.
	
	The `-Aa` 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:
	
	`#define _POSIX_SOURCE`
	
	or define this macro by using the `-D` *name* option:
	
	`cc -D _POSIX_SOURCE  myfile.c`
	
	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:
	
	`#define _XOPEN_SOURCE`

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does

-A*level* (continued)	or define the macro on the command line:
	`cc -D _XOPEN_SOURCE myfile.c`
	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:
	`#define _HPUX_SOURCE`
	or define the macro on the command line:
	`cc -D _HPUX_SOURCE myfile.c`
	`_HPUX_SOURCE` defines a superset of names defined by `_XOPEN_SOURCE`. `_XOPEN_SOURCE` defines a superset of names defined by `_POSIX_SOURCE`.

c	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.
e	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 at the HP-UX 10.30 and later operating system releases.
-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 *HP-UX Reference* manual for details.
-D*name* -D*name*=*def*	Defines *name* to the preprocessor `cpp` as if defined by the preprocessing directive `#define`. If =*def* is not given, *name* 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 `#if` and `#else`. See the section titled "Conditional Compilation" in Chapter 7 for more information.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`-E`	Runs the preprocessor only on the named HP C or assembly programs and sends the result to standard output (`stdout`).
`-g`	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: ■ 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 To enable debugging of optimized code, specify the `-g` compile-line option together with the `-O`, `+O1`, or `+O2` option. Currently, debugging is supported at optimization levels 2 and below. If you try to use `-g` with the `+O3` or `+O4` option, the compiler will issue a warning stating that the options are incompatible, and will ignore the `-g` option.
`-G`	Prepares the object file for profiling with `gprof`. See the `gprof(1)` description in the *HP-UX Reference* manual for details.
`-I`*dir*	Adds *dir* 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 `/`, the preprocessor first searches the current directory, then the directory named in the `-I` option, and finally, the standard include directory `/usr/include`. For include files that are enclosed in `<` and `>` symbols, 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.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`-l`*x*	Causes the linker to search the libraries `/usr/lib/lib`*x*`.a` or `/usr/lib/lib`*x*`.sl` (searched first) and `/opt/langtools/lib/lib`*x*`.a` or `/opt/langtools/lib/lib`*x*`.sl` (searched second). The `-a` linker option determines whether the archive (`.a`) or shared (`.sl`) version of a library is searched. The linker searches the shared version of a library by default. Libraries are searched when their names are encountered. Therefore placement of a `-l` is significant. If a file contains an unresolved external reference, the library containing the definition must be placed after the file on the command line. See the description of `ld(1)` in the *HP-UX Reference* for details.
`-L`*dir*	Causes the linker to search for libraries in the directory *dir* before using the default search path. The default search path is the directory `/usr/lib` followed by `/opt/langtools/lib`. `-L`*dir* must precede `-l`*x* on the command line; otherwise `-L`*dir* is ignored. This option is passed directly to the linker. For example: `cc -L/project/libs prog.c -lfoo -lbar` Compiles and links `prog.c` and directs the linker to search the directories `/project/libs`, `/usr/lib`, then `/opt/langtools/lib` for libfoo.a, libfoo.sl, libbar.a, and libbar.sl libraries.
`-n`	Causes the program file produced by the linker to be marked as shareable. For details and system defaults see the description of `ld(1)` in the *HP-UX Reference* manual.
`-N`	Causes the program file produced by the linker to be marked as unshareable. For details and system defaults see the `ld(1)` description in the *HP-UX Reference* manual.
`-o` *outfile*	Causes the output of the compilation sequence to be placed in *outfile*. The default name is `a.out`. 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.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`-O`	Invokes the optimizer to perform level 2 optimization. Other optimization levels can be set. Refer to the *HP C Programmer's Guide* 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 `monitor(3C)` at the start and writes out a `mon.out` file upon normal termination of the object program's execution. An execution profile can then be generated using `prof(1)`.
`-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` as the suffix.
`-q`	Causes the output file from the linker to be marked as demand loadable. For details and system defaults, see the `ld(1)` description in the *HP-UX Reference* manual.
`-Q`	Causes the program file created by the linker to be marked as *not* demand loadable. For details and system defaults, see the `ld(1)` description in the *HP-UX Reference* 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 `ld(1)` description in the *HP-UX Reference* manual for more details.
`-S`	Compiles the named HP C program and leaves the assembly language output in a corresponding file with `.s` as the suffix.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`-t` *x,name*	Substitutes or inserts subprocess *x* with *name*, where *x* is one or more of a set of identifiers indicating the subprocesses. This option works in two modes: 1) if *x* is a single identifier, *name* represents the full pathname of the new subprocess; 2) if *x* is a set of identifiers, *name* represents a prefix to which the standard suffixes are concatenated to construct the full pathnames of the new subprocesses. *x* can be one or more of the following values: **Value** — **Description** `p` — Preprocessor (standard suffix is `cpp`). `c` — Compiler (standard suffix is `ccom`). `0` — Same as `c`. `a` — Assembler (standard suffix is `as`). `l` — Linker (standard suffix is `ldr`).
`-U`*name*	Undefines or removes any initial definition of *name* in the preprocessor. See the `cpp(1)` description in the *HP-UX Reference* manual for details.
`-v`	Enables the verbose mode sending a step-by-step description of the compilation process to standard error (`stderr.`)
`-V`	Prints version information on each invoked subprocess to standard error (`stderr.`)
`-w`	Suppresses warning messages.
`-W`*x*, *arg1* *[,arg2,..,argn]*	Passes the arguments *arg1* through *argn* to the subprocess *x* of the compilation. *x* can be one of the values described under the `-t` option with the addition of `d`, to pass an option to the cc driver. These options are HP value added extensions to HP C.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
-W c,-e or +e	Enables the following HP value added features while compiling in ANSI C mode: sized enum, long long, long pointers, compiler supplied defaults for missing arguments to intrinsic calls, and $ in identifier HP C extensions.
-W c,-L or +L	Enables any #pragma listing directives and the listing facility. A straight listing prints the following: ■ 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). 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.
-W c,-m or +m	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.
-W c,-o or +o	Causes the code offsets to be printed in hexadecimal grouped by function at the end of the listing.
-W c,-R*num* or +R*num*	Allow only the first *num* register variables to actually have the register class. Use this option when the register allocator issues an *out of general registers* message.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`-W c,-w`*n* or `+w`*n*	Specifies the level of the warnings messages. The value of *n* can be one of the following: **Value** — **Description** `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.
`-W d,-a`	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. 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 *HP C Programmer's Guide*.
`-y`	Requests the compiler to generate additional information for the static analysis tool, which is a part of the *HP SoftBench* software development environment. See *HP Softbench Static Analyzer: Analyzing Program Structure* for more information.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`-Y`	Enables Native Language Support (NLS) of 8-bit and 16-bit characters in comments, string literals, and character constants. See `hpnls(5)`, `long(5)`, and `environ(5)` in the *HP-UX Reference* 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" in Chapter 10 for a description of this path.
`-z`	Disallows run-time dereferencing of null pointers. Fatal errors result if null pointers are dereferenced.
`-Z`	Allows dereferencing of null pointers at run-time. (This behavior is the default.) The value of a dereferenced null pointer is zero.
`+DA`*model*	Generates object code for a particular version of the PA-RISC architecture. If you do not specify this option, the *default* object code generated is determined automatically as that of the system on which you run the compilation. `+DA` also specifies which version of the HP-UX math library to link in when you have specified `-lm`. (See the *HP-UX Floating-Point Guide* for more information about using math libraries.) *model* 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`, or `2.0`, or `portable`. For example, specifying `+DA1.1` or `+DA735` generates code for the PA-RISC 1.1 architecture. Similarly, specifying `+DA2.0` generates code for the PA-RISC 2.0 architecture. Specifying `+DAportable` generates code compatible across HP 9000 workstations and servers. See the file `/opt/langtools/lib/sched.models` for model numbers and their architectures. Use the command `uname -m` to determine the model number of your system.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`+DA`*model* (continued)	Note: Object code generated for PA-RISC 2.0 will *not* execute on PA-RISC 1.1 systems. To generate code compatible across PA-RISC 1.1 and 2.0 workstations and servers, use the `+DAportable` option. 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.) For more information on this option, see "Using +DA to Generate Code for a Specific Version of PA-RISC" in this chapter.
`+df`*name*	Specifies the path name of the profile database to use with profile-based optimization. This option can be used with the `+P` command line option. The profile database by default is named `flow.data`. This file stores profile information for one or more executables. Use `+df` when the `flow.data` file has been renamed or is in a different directory than where you are linking. You can also use the `FLOW_DATA` environment variable to specify a different path and file name for the profile database file. The `+df`*name* command line option takes precedence over the `FLOW_DATA` environment variable. Note, the `+df`*name* option cannot be used to redirect the instrumentation output (with the `+I` option). It is only compatible with the `+P` option.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`+DS`*model*	Performs instruction scheduling tuned for a particular implementation of the PA-RISC architecture. *model* 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. To obtain the best performance on a particular model or processor of the HP 9000, use `+DS` with that model number or processor name. If you plan to run your program on both PA-RISC 1.1 and 2.0 systems, use the +DS2.0 designation. 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 `uname -m` to determine the model number of your system. Object code with scheduling tuned for a particular model or processor *will* execute on other HP 9000 systems, although possibly less efficiently. For more information on this option, see "Using +DS to Specify Instruction Scheduling" in this chapter.
`+e`	See `-W c,-e`
`+ESfic`	Replaces millicode calls with inline fast indirect calls. The `+ESfic` compiler option affects how function pointers are dereferenced in generated code. The default is to generate low-level millicode calls for function pointer calls.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`+ESfic` (continued)	The `+ESfic` option generates code that calls function pointers directly, by branching through them. The `+ESfic` option should only be used in an environment where there are no dependencies on shared libraries. The application must be linked with archive libraries. Using this option can improve run-time performance.

`+ESlit`	Places string literals and constants defined with the ANSI C `const` type qualifier into the `$LIT$` subspace. The `$LIT$` subspace is used for read-only data storage. This option can reduce memory requirements and improve run-time speed in multi-user applications.
	
	Normally the C compiler only places floating-point constant values in the `$LIT$` subspace, and other constants and literals in the `$DATA$` subspace. This option allows the placement of large data objects, such as ANSI C `const` arrays, into the `$LIT$` subspace.
	
	All users of an application share the static data stored in the `$LIT$` subspace. Each user is allocated a private copy of the dynamic data stored in the `$DATA$` subspace. By moving additional static data from the `$DATA$` subspace to the `$LIT$` subspace, overall system memory requirements can be reduced and run-time speed improved. Most applications can benefit from this option.
	
	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.
	
	`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 *);`

Table 9-2. HP C Compiler Option Details (continued)

<table>
<tr><th>Option</th><th>What It Does</th></tr>
<tr><td><code>+ESnoparmreloc</code></td><td><code>+ESnoparmreloc</code> 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.

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.

For example, <code>cos</code> has not been compiled with <code>+ESnoparmreloc</code>. If you declare <code>cos</code> with a function prototype as shown below, and compile with <code>+ESnoparmreloc</code>, an incorrect value will be passed to <code>cos()</code>.

<pre>double cos(double);
main()
        {
        printf("%f\n", cos(1.0));
        }</pre>For more information on parameter relocation, see the HP PA-RISC Procedure Calling Conventions Reference Manual, HP part number 09740-90015.</td></tr>
<tr><td><code>+ESsfc</code></td><td>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.

The <code>+ESsfc</code> option generates code that compares function pointers directly, as if they were simple integers.

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.</td></tr>
</table>

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`+f`	Inhibits the automatic promotion of `float` to `double` when evaluating expressions. This differs from the `+r` option in that both parameters *and* function return values are still promoted to `double`. In ANSI mode, this option is ignored and a warning is issued.
`+FP`*flags*	Specifies what floating-point traps to enable and also enables or disables fast underflow mode. *flags* 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 *flags* are: `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.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`+FP`*flags* (continued)	`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.) `d` Disable fast underflow (flush to zero) of denormalized values. To dynamically change these settings at run time, refer to *fpgetround(3M)*.
`+help`	Invokes the initial menu window of the online programmer's guide. If `+help` is used on any command line, the compiler invokes the online programmer's guide and then processes any other arguments. If `$DISPLAY` is set, `+help` invokes the `helpview`(1X) command. If the display variable is not set, a message so indicates. For example, the following command: `cc -c sub.c prog.c +help` invokes the online programmer's guide and compiles `sub.c` and `prog.c` into relocatable object code files.
`+I`	Instructs the compiler to instrument the object code for collecting run-time profile data. The profiling information can then be used by the linker to perform profile-based optimization. Code generation and optimization phases are delayed until link time by this option. After compiling and linking with `+I`, run the resultant program using representative input data to collect execution profile data. Profile data is stored in `flow.data` by default. See the `+df`*name* option for information on controlling the name and location of this data file.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`+k`	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. 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. 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.
`+L`	See `-W c,-L`
`+m`	See `-W c,-m`
`+M`	Emit warnings of Quiet Changes, use the `+M` option with either `-Aa` or `-Ac`. Warnings of Quiet Changes were formerly part of `+w1`. The document *Rationale for Draft Proposed American National Standard for Information Systems* available with the *ANSI Programming Language C Standard ISO 9899:1990* discusses the behavioral differences in code that is migrated to ANSI C. These differences are called Quiet Changes.
`+o`	See `-W c,-o`

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`+O`*opt*	Invokes the level of optimization selected by *opt*. *opt* can have any of the following values. `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. For a complete list of advanced optimization toggles, see Chapter 4 "Optimizing HP C Programs" of the *HP C Programmer's Guide*.
`+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. The `+P` option does not affect the default optimization level, or the optimization level specified by the `-O` or `+O`*opt* 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. The `+P` command line option is incompatible with the `+I`, `-g`, `-s`, `-S`, and `-y` options. Also refer to the `+I`, `+pgm` and `+df` command line options. For more information on using these options, see the *HP C Programmer's Guide*.
`+pgm`*name*	Specifies a program name to look up in the **`flow.data`** file. Used with profile-based optimization and the `+P` option. `+pgm`*name* should be used when the name of the profiled executable differs from the name of the current executable specified by the `-o` option.

Table 9-2. HP C Compiler Option Details (continued)

Option	What It Does
`+R`*num*	See `-W c, -R`*num*
`+r`	Inhibits the automatic promotion of `float` to `double` when evaluating expressions and passing arguments. This option is not valid in ANSI mode.
`+u`*bytes*	Forces all pointer references to assume that data is aligned on either an 8-bit, 16-bit, or 32-bit addresses. *bytes* 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 *HP C Programmers Guide* for more information.
`+w`*n*	See `-W c,-w`*n*
`+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. The `-p` and `-G` options are incompatible with this option and are ignored. See the *HP-UX Linker and Libraries Online User Guide* 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. For more information on shared libraries, see the *HP-UX Linker and Libraries Online User Guide*.

Any other option encountered generates a warning to standard error (`stderr`.) (Options not recognized by `cc` are *not* 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`.

- `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 +w1 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.

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
```

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 /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 `+DA`*model* with the model number of the target system.

  For example, if you are compiling on a 720 and your program will run on an 855, you should use `+DA855`. 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 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, 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.)

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, 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, 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 `+DS`*model* 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 `+DS`*model* 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:

- intrinsic pragmas
- copyright notice and identification pragmas
- data alignment pragmas
- optimization pragmas
- program listing pragmas
- shared library pragmas

Refer to the *HP C Programmer's Guide* for additional information on pragmas.

Intrinsic Pragmas

See Chapter 11, "Using Intrinsics" 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.

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.

Optimization Pragmas

For additional information on the following optimization pragmas see Chapter 4, "Optimizing C Programs," of the *HP C Programmer's Guide*.

ALLOCS_NEW_MEMORY Pragma

#pragma ALLOCS_NEW_MEMORY $functionname_1$,...,$functionname_n$

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 Programmer's Guide* for additional information.)

FLOAT_TRAPS_ON Pragma

#pragma FLOAT_TRAPS_ON { *functionname*, ... *functionname* | _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 [$functionname_1$,...,$functionname_n$]

#pragma NOINLINE [$functionname_1$,...,$functionname_n$]

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 *C 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 *C Programmer's Guide* for details and examples.

NO_SIDE_EFFECTS Pragma

`#pragma NO_SIDE_EFFECTS` $functionname_1$`,...,`$functionname_n$

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 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.

Data Alignment Pragma

This section discusses the data alignment pragma `HP_ALIGN` and its 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 Programmer's Guide* for detailed information on the `HP_ALIGN` pragma.

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 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 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 +u*bytes* 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 +u*bytes* 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 +u*number* 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 `+u`*bytes* option, see Table 9-2.

Listing Pragmas

The listing pragmas introduced in this section are discussed in more detail in Chapter 12.

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  }
             { 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  }
                 { 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.

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).

10

HP C/HP-UX Implementation Topics

This chapter describes topics that are specific to programming in C on HP 9000 workstations and servers.

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. More are honored when the `+R` option is used.
- 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 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`	32	4	-2,147,483,648	2,147,483,647	Signed integer
`long long`	64	8	-2^{63}	2^{63} -1	Signed integer
`unsigned long`	32	4	0	4,294,967,295	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.

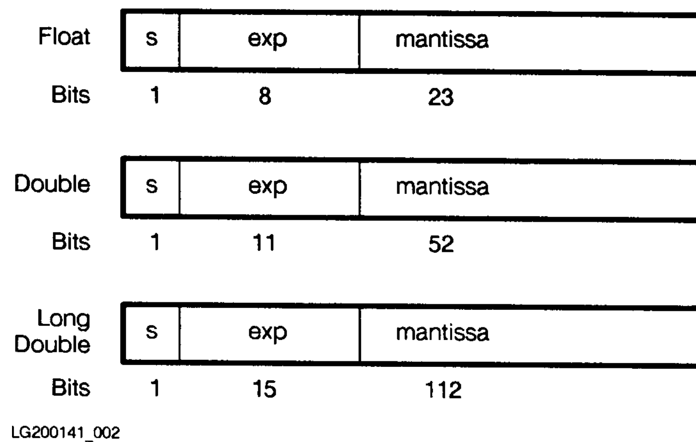

Figure 10-1. Internal Representation of Floating-Point Numbers

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:

	float	**double**	**long double**
bias	+127	+1023	+16383
$\mathbf{E_{max}}$	+127	+1023	+16383
$\mathbf{E_{min}}$	-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

$E_{min} <= E <= E_{max}$

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 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;

    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);
}
```

HP Specific Type Qualifiers

See the section "HP Specific Type Qualifiers" in chapter 3.

Location of Files

Table 10-2 lists the location of the HP C files.

Table 10-2. 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
Advanced Optimizing Code Generator	/usr/ccs/lbin/ucomp
C libraries (libc)	/usr/lib/libc.a[1] /lib/libc.sl
Debugging Aid Routines	/opt/langtools/lib/end.o
man Pages	/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)

Table 10-2. Location of Files (continued)

File or Library	Location
Message Catalogs	/opt/ansic/lib/nls/msg/C/cc.cat (English)
	/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

Table 10-2. Location of Files (continued)

File or Library	Location
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]
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]

Table 10-2. Location of Files (continued)

File or Library	Location
Parallel Runtime Library	/opt/langtools/lib/libmp.a
Standard Include Files	/usr/include
Online Help	/opt/ansic/dt/appconfig/help/C
C Tools	/opt/ansic/bin
	/opt/ansic/lbin
	/opt/langtools/bin
	/opt/langtools/lbin

[1]You can change the search path for `libc.a` through the `-L`*dir* option. See Chapter 9 for details.

[2]See the *HP-UX Linker and Libraries Online User Guide* for more information on PBO of shared libraries.

[3]You 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.

[4]The appropriate library file is automatically selected based upon the options selected on the command line.

[5]The `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.

[6]For 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*.

11

Using Intrinsics

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, in turn, 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  }
                 { OFF }
```

Causes a page break after each function definition. Default is `OFF`.

```
#pragma LIST { ON  }
             { 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:

```
auto          external definition        static
constant      member                     typedef
external      register
```

The third column shows the type of the identifier. The types include:

```
array         int                  unsigned char
char          long int             unsigned int
```

double	long long int	unsigned long
enum	short int	unsigned long long
float	struct	unsigned short
function	union	void

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;

}


       G L O B A L    I D E N T I F I E R    M A P

Identifier          Class       Type              Address
----------          -----       ----              -------
 main               ext def     int ()            main
```

```
        L O C A L   I D E N T I F I E R   M A P S


                         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
  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		

A

Syntax Summary

This appendix presents a summary of the C language syntax as described in this manual.

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 ::=

+
-

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*

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]

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*
    `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 newline*

`# 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`

Index

A

-Aa option, 9-8
-Ac compiler option, 9-10
\a (control G escape code), 2-20
addition operator (+), 5-22
address-of operator (&), 5-16
adjacent character string literals, 2-23
adjacent wide string literals, 2-23
-Ae compiler option, 9-10
aggregate
  initializing, 3-32
  types, 2-9
ALLOCS_NEW_MEMORY pragma, 9-35
AND bitwise operator (&), 5-29
AND logical operator (&&), 5-32
ANSI C
  mode, 1-2, 9-2
  standard, 1-1
ANSI migration warnings, 9-24
a.out file, 9-3
apostrophes, 2-18
arithmetic
  conversions, 4-5
  operators, 5-17
  types, 2-9
array, 2-10, 4-7
  declarator, 3-25
  maximum number of dimensions, 10-8
  row-major storage, 3-25
  storage, 3-25
  subscripting, 5-7
assembly source files, 9-2
assignment, 4-1
  conversion, 11-2
  expression, 5-4, 5-37
  operator (=), 5-36
automatic storage duration, 2-7
AUTOPAGE pragma, 9-42, 12-2
auto scalar objects, 3-32
auto storage class specifier, 3-5

B

\\(backslash escape code), 2-20
\b (backspace escape code), 2-20
bit-fields, 3-15, 4-5, 10-3
  manipulation, 5-29
bitwise AND operator (&), 5-29
bitwise exclusive OR operator (^), 5-30
bitwise inclusive OR operator (|), 5-31
bitwise shift operators, 5-24, 10-7
block of code, 6-3
block scope, 2-4
break statement, 6-21

C

calling a function, 5-9
  by reference, 5-11
  by value, 5-11
casting, 4-1
cast operator, 3-28, 5-19
cc command, 9-1
-c compiler option, 9-10

-C compiler option, 9-10
CCOPTS environment variable, 9-29
character constants, 2-18
  integral, 2-19
  wide, 2-19
characters, 2-22
  string literals, 2-23
char type, 3-7
C libraries, 9-12
code offsets, 9-24, 12-7
comma operator (,), 5-39
comments in HP C programs, 2-26
compilation
  ANSI mode, 1-2
  compatibility mode, 1-2, 9-2
  compilation process, 9-1
  compiler options, 9-8
  conditional, 7-1, 7-11
  PA-RISC architecture versions, 9-30
compiler options, 9-8
  -Aa, 9-8
  -Ac, 9-10
  -Ae, 9-10
  -c, 9-10
  -C, 9-10, 12-1
  +DA*model*, 9-17, 9-30
  +df*name*, 9-18
  -D*name*, 9-10
  +DS*model*, 9-19, 9-31
  +e, 9-14
  -E, 9-11
  +ESfic, 9-19
  +ESlit, 9-20
  +ESnoparmreloc, 9-21
  +ESsfc, 9-21
  +f, 9-21
  +FP*flags*, 9-22
  -g, 9-11
  -G, 9-11
  +help, 9-23
  +I, 9-23
  -I*dir*, 9-11
  +k, 9-24
  +L, 9-15, 9-24, 12-1
  -l, 9-12
  -L*dir*, 9-12
  +m, 9-15, 9-24, 12-3
  +M, 9-24
  -n, 9-12
  -N, 9-12
  +o, 9-15, 9-24
  -o, 9-12
  -O, 9-13
  +O*opt*, 9-25
  +P, 9-25
  -p, 9-13
  -P, 9-13
  +pgm*name*, 9-25
  -q, 9-13
  -Q, 9-13
  +r, 9-26
  +R*num*, 9-15, 9-26
  -s, 9-13
  -S, 9-13
  summary, 9-4
  -t, 9-14
  +u*bytes*, 9-26, 9-39
  -U*name*, 9-14
  -v, 9-14
  -V, 9-14
  -w, 9-14
  -Wc,-0, 12-3
  -Wc,-e, 9-14
  -Wc,-L, 9-15
  -Wc,-m, 9-15, 12-3
  -Wc,-o, 9-15, 12-7
  -Wc,-R*num*, 9-15
  -Wc,-w*n*, 9-15
  -Wd, -a, 9-16
  +w*n*, 9-15, 9-26
  -W*x*, 9-14
  -Y, 9-17

-y option, 9-16
+z, 9-26
+Z, 9-26
-z, 9-17
-Z, 9-17
compiling HP C programs, 9-1
compound statement, 6-3
conditional compilation, 7-1, 7-2, 7-11
conditional operator (?:), 5-34
constant, 2-12
character, 2-18
decimal, 2-14
defined, 2-12
enumeration, 2-18
expression, 5-18, 5-40
floating, 2-12
hexadecimal, 2-14
integer, 2-14
octal, 2-14
continuation character, 2-1
continue statement, 6-20
conversions
arithmetic, 4-5
data type, 4-1
floating, 4-6
integral, 4-5
COPYRIGHT_DATE pragma, 9-34
COPYRIGHT pragma, 9-34
cpp(1), 9-2, 9-10

D

+DA*model* compiler option, 9-17, 9-30
data alignment pragma, 9-37
data declarations, 10-8
data representation, 4-5
data type, 3-3
ranges, 10-1
sizes, 10-1
data type conversion, 4-1
data types, 10-1
aggregate, 2-9
arithmetic, 2-9
as implemented in HP C/HP-UX, 10-1
char, 3-7
double, 3-7
enumeration, 2-9
float, 3-7
floating-point, 2-9
function, 2-9
int, 3-7
integral, 2-9
long, 3-7
long long, 3-7
pointer, 2-9
scalar, 2-9
short, 3-7
structure, 2-9
union, 2-9
void, 2-9
decimal constant, 2-14
declaration, 3-3
declarator, 3-23
array, 3-25
pointer, 3-24
decrement operator (--), 5-6, 5-13
decrement operator (—), 5-15
default listing, 12-1
defined operator, 7-11
demand loadable, 9-13
digit, 2-3, 2-14
directives, preprocessor, 7-1
division operator (/), 5-20
-D*name* compiler option, 9-10
do statement, 6-14
\" (double quote escape code), 2-20
double type, 3-7
+DS*model* compiler option, 9-19, 9-31

E

+e compiler option, 9-14
-E compiler option, 9-11

#elif, 7-11
#else, 7-11
else clause, 6-7
#endif, 7-11
enum
  declaration, 3-20
  statement, 2-18
enumeration, 3-20
  constants, 2-18
  types, 2-9
enumerator, 3-20
environment variables, 9-29
  CCOPTS, 9-29
  TMPDIR, 9-30
equality operators, 5-27
#error, 7-17
error messages, 1-3
escape sequences
  hexadecimal, 2-20
  octal, 2-20
+ESfic compiler option, 9-19
+ESlit compiler option, 9-20
+ESnoparmreloc compiler option, 9-21
+ESsfc compiler option, 9-21
expression, 5-1, 6-5
  assignment, 5-37
  constant, 5-40
extern, 6-3
  declarations, 10-8
  storage class specifier, 3-5

F

+f compiler option, 9-21
`\f` (form feed escape code), 2-20
file
  scope, 2-4
floating-point
  constants, 2-12
  conversions, 4-6
  operations, 9-22
  types, 2-9, 3-7
float type, 3-7
for statement, 6-15
+FP*flags* compiler option, 9-22
function, 4-7, 10-10
  call, 4-1, 5-9
  call by reference, 5-11
  call by value, 5-11
  declarator, 3-26
  definitions, 3-37
  library, 8-2
  prototypes, 2-4, 3-37
  referencing functions instead of macros, 8-2
  scope, 2-4
function types, 2-9

G

-g compiler option, 9-11
-G compiler option, 9-11
goto statement, 6-2, 6-19
gprof(1), 9-11

H

header files
  location, 8-1
  specifying, 8-1
+help compiler option, 9-23
help, online, 1-3
hexadecimal
  constants, 2-14
  escape sequences, 2-20
HP_ALIGN pragma, 9-37, 9-38, 9-39
HP C file location, 10-14
HP C source files, 9-2
HP_SHLIB_VERSION pragma, 9-36
HP specific type qualifier, 3-9, 10-13
_HPUX_SOURCE name space macro, 9-10

I

+I compiler option, 9-23

identifier, 2-3, 2-5, 2-8
  maps, 9-24, 12-3
  scope, 2-4
  types, 2-8
-I*dir* compiler option, 9-11
IEEE floating-point format, 10-4
#if, 7-11
#ifdef, 7-11
#ifndef, 7-11
if statement, 6-7
include files, 7-1
inclusive OR operator , 5-31
increment operator (++), 5-6, 5-13, 5-15
indirection operator, 5-16
initialization
  aggregates, 3-32
  auto scalar objects, 3-32
  expression, 6-15
  static objects, 3-32
initialization of objects, 3-32
integer
  constants, 2-14
  types, 2-9
integral
  character constants, 2-19
  conversions, 4-5
  promotion, 4-2
integral promotion
  unsigned preserving rules, 4-2
integral types, 2-9
intrinsic, 11-1, 11-2
  defined, 11-1
INTRINSIC_FILE pragma, 9-33, 11-4
INTRINSIC pragma, 9-33, 11-2
int type, 3-7
iteration statements, 6-11

J

jump statements, 6-17

K

+k compiler option, 9-24
keywords, 2-2
  listed, 2-2

L

labeled statements, 6-2
labels, 2-8, 6-2
language support, 9-17
+L compiler option, 9-15, 9-24, 12-1
-L*dir* compiler option, 9-12
left shift operator (<<), 5-24
lexical elements, 2-1
library functions, 8-2
#line, 7-15
line control, 7-15
line number specification, 7-1
LINES pragma, 9-41, 12-2
linkage of an identifier, 2-5
linking, 9-1
listing, 9-24
  facility, 12-1
  format, 12-1
  options, 12-3
  pragmas, 9-41, 12-2
LIST pragma, 9-42, 12-2
literal
  string, 2-22
  wide string, 2-23
LOCALITY pragma, 9-34
location of HP C files, 10-14
  table, 10-14
logical AND operator (&&), 5-32
logical complement, 5-17
logical OR operator (||), 5-33
long long type, 2-15, 3-7
long type, 3-7
loop body, 6-11
lvalue, 5-4
-l*x* compiler option, 9-12

M

macro, 2-8, 10-10
  definition, 7-6
  names, 2-4
  predefined, 7-10
  replacement, 7-6
marco
  replacement, 7-2
math libraries, 10-9
  location of files, 10-16
maximum number of array dimensions, 10-8
+m compiler option, 9-15, 9-24, 12-3
+M compiler option, 9-24
multiplication operator (*), 5-20
multiplicative operators, 5-20

N

names
  type, 3-28
name spaces, 2-5, 2-8, 9-10
Native Language Support (NLS), 9-17
-n compiler option, 9-12
-N compiler option, 9-12
nesting level, 9-24
NLS, 9-17
`\n` (newline character escape code), 2-20
[NO]INLINE pragma, 9-35
nondigit, 2-3
nonprinting characters, 2-20
no-operation statements, 6-5
[NO]PTRS_STRONGLY_TYPED pragma, 9-36
NO_SIDE_EFFECTS pragma, 9-36
null statement, 6-5

O

+o compiler option, 9-15, 9-24
-o compiler option, 9-12
-O compiler option, 9-13
octal constant, 2-14
octal escape sequences, 2-20
ones complement, 5-17
online help, 1-3
+O*opt* compiler option, 9-25
operator, 2-24, 2-25
  (--), 5-13
  addition (+), 5-22
  address-of (&), 5-16
  assignment, 5-36
  bitwise AND (&), 5-29
  bitwise exclusive OR (^), 5-30
  bitwise inclusive OR (|), 5-31
  bitwise shift, 5-24
  cast, 5-19
  comma (,), 5-39
  conditional (?:), 5-34
  defined, 7-11
  division (/), 5-20
  equality, 5-27
  indirection, 5-16
  logical AND (&&), 5-32
  logical OR (||), 5-33
  multiplication (*), 5-20
  multiplicative, 5-20
  postfix, 5-6
  postfix decrement, 5-13
  postfix increment, 5-13
  precedence, 5-2
  precedence table, 5-3
  prefix decrement, 5-15
  prefix increment, 5-15
  relational, 5-25
  remainder (%), 5-20
  sizeof, 5-18
  subtraction (-), 5-22
  tokens, 2-24
  unary, 5-14
  unary arithmetic, 5-17
optimization, 9-1, 9-25
  optimizer, 9-13

OPTIMIZE pragma, 9-35
OR (bitwise exclusive) operator (^), 5-30
OR (bitwise inclusive) operator (|) , 5-31
OR (logical) operator (||), 5-33
overflow expression, 10-7
overloading classes, 2-8

P

PAGE pragma, 9-42, 12-2
parallel runtime library, 10-17
PA-RISC architecture versions, 9-30
+P compiler option, 9-25
-p compiler option, 9-13
-P compiler option, 9-13
+pgm*name* option, 9-25
pointer, 4-7, 10-7
  declarator, 3-24
pointer types, 2-9
portability, 1-1
_POSIX_SOURCE name space macro, 9-9, 9-10
postfix
  decrement operator, 5-13
  increment operator, 5-13
  operator defined, 5-6
#pragma, 7-16
pragma
  [NO]PTRS_STRONGLY_TYPED, 9-36
pragmas, 7-1, 9-33
  ALLOCS_NEW_MEMORY, 9-35
  AUTOPAGE, 9-42, 12-2
  COPYRIGHT, 9-34
  COPYRIGHT_DATE, 9-34
  data alignment, 9-37
  HP_ALIGN DOMAIN_NATURAL, 9-38
  HP_ALIGN DOMAIN_WORD, 9-38
  HP_ALIGN HPUX_NATURAL, 9-38
  HP_ALIGN HPUX_NATURAL_S500, 9-38
  HP_ALIGN HPUX_WORD, 9-38
  HP_ALIGN NATURAL, 9-38
  HP_SHLIB_VERSION, 9-36
  INTRINSIC, 9-33, 11-2
  INTRINSIC_FILE, 9-33, 11-4
  LINES, 9-41, 12-2
  LIST, 9-42, 12-2
  LOCALITY, 9-34
  [NO]INLINE, 9-35
  NO_SIDE_EFFECTS, 9-36
  OPTIMIZE, 9-35
  PAGE, 9-42, 12-2
  SUBTITLE, 9-42, 12-2
  TITLE, 9-42, 12-2
  VERSIONID, 9-35
  WIDTH, 9-41, 12-2
precedence of operators, 5-2
predefined macros
  `__DATE__`, 7-10
  `__FILE__`, 7-10
  `__LINE__`, 7-10
  `__STDC__`, 7-10
  `__TIME__`, 7-10
prefix decrement operator, 5-15
prefix increment operator, 5-15
preprocessing directives
  #define, 7-6
  #elif, 7-11
  #else, 7-11
  #endif, 7-11
  #error, 7-17
  #if, 7-11
  #ifdef, 7-11
  #ifndef, 7-11
  #include, 7-4
  #line, 7-15
  #pragma, 7-16
preprocessor, 9-1, 9-10, 9-13
preprocessor directives, 7-1

primary expression, 5-5
profile-based optimization, 9-25
profiling, 9-11
promoting floating-point numbers as floats, 9-21
promotion, integral, 4-2
punctuators, 2-25

Q

-q compiler option, 9-13
-Q compiler option, 9-13
\? (question mark escape code), 2-20
Quiet Changes, 9-24

R

ranges of data types, 10-1
\r (carriage return escape code), 2-20
+r compiler option, 9-26
register storage class specifier, 3-5
register variables, 9-26
relational operators, 5-25
relocatable object file, 3-2, 9-2
remainder operator (%), 5-20
reserved words, 2-2
returned values, 4-1
return statement, 6-22
right shift operator (>>), 5-24
+R*num* compiler option, 9-15, 9-26
row-major array storage, 3-25
running HP C programs, 9-43

S

scalar objects
  initializing, 3-32
scalar types, 2-9
-s compiler option, 9-13
-S compiler option, 9-13
scope, 10-8
  of an identifier, 2-4
  of function prototypes, 2-4
selection statements, 6-6
self-referential structure, 3-17
shared libraries, 9-1
  shared library pragma, 9-36
short type, 3-7
simple assignment, 5-36
\' (single quote escape code), 2-20
sized enum, 3-21
sizeof operator, 3-28, 5-18
sizes of data types, 10-1
source files
  assembly, 9-2
  HP C, 9-2
  inclusion, 7-4
special characters table, 2-20
specifier
  storage-class, 3-5
  structure, 3-14
  type, 3-7
  union, 3-14
statements, 5-1
  break, 6-21
  compound, 6-3
  continue, 6-20
  defined, 6-1
  do, 6-14
  for, 6-15
  goto, 6-19
  groups, 6-1
  if, 6-7
  iteration, 6-11
  jump, 6-17
  labeled, 6-2
  no-operation, 6-5
  null, 6-5
  return, 6-22
  selection, 6-6
  switch, 6-9
  while, 6-13
static, 6-3
  objects, 3-32
  storage duration, 2-7

static storage class specifier, 3-5
<stdarg.h>, 10-10
storage-class specifiers, 3-5
storage duration, 2-7
string
  literal, 2-22
structure, 3-14, 10-6
  members, 5-12
  self-referential, 3-17
  specifier, 3-14
  tag, 3-16
structure types, 2-9
structure/union member (.), 5-6, 5-12
structure/union pointer (->), 5-6
structure/union pointer (arrow), 5-12
subprocess, 9-14
subscript operator ([ ]), 5-6
SUBTITLE pragma, 9-42, 12-2
subtraction operator (-), 5-22
switch statement, 6-2, 6-9
symbolic debugger, 9-11
symbol table, 9-13

T

tag, 2-8
  structure, 3-16
  union, 3-16
-t compiler option, 9-14
`\t` (horizontal tab escape code), 2-20
thread HP specific type qualifier, 3-9
TITLE pragma, 9-42, 12-2
TMPDIR environment variable, 9-30
token, 5-1
  defined, 2-1
  operator, 2-24
  syntax, 2-1
translation unit, 3-2
trigraph sequences, 7-18
type
  const qualifier, 3-11
  conversions, 4-1
  definitions, 3-30
  enumeration, 3-20
  mismatches in external names, 10-7
  names, 3-28
  specifiers, 3-7
  volatile qualifier, 3-11
typedef keyword, 3-5, 3-30
types of identifiers, 2-8

U

+u*bytes* compiler option, 9-26, 9-39
-U*name* compiler option, 9-14
unary arithmetic operators, 5-17
unary operators, 5-14
underflow expression, 10-7
union, 3-15, 10-6
  members, 5-12
  specifier, 3-14
  tag, 3-16
union types, 2-9
unsigned keyword, 3-7
/usr/lib/sysintr , 11-4

V

va_args macros, 10-10
va_dcl macro, 10-10
value preserving rules, 4-2
va_start macro, 10-10
-v compiler option, 9-14
-V compiler option, 9-14
verbose mode, 9-14
VERSIONID pragma, 9-35
version print, 9-14
void type, 3-7
void types, 2-9
volatile type qualifier, 3-11
`\v` (vertical tab escape code), 2-20

W

-Wc,-e compiler option, 9-14
wchar_t typedef, 2-20, 2-23, 3-34

-Wc,-L compiler option, 9-15
-Wc,-m compiler option, 9-15
-Wc,-o compiler option, 9-15
-w compiler option, 9-14
-Wc,-R*num* compiler option, 9-15
-Wc,-w*n* compiler option, 9-15
-Wd, -a compiler option, 9-16
while statement, 6-13
white space, 2-1
wide character constants, 2-19
wide string literal, 2-23
WIDTH pragma, 9-41, 12-2
+w*n* compiler option, 9-15, 9-26
-W*x* compiler option, 9-14

X

_XOPEN_SOURCE name space macro,
9-9, 9-10
XOR, 5-30

Y

-y compiler option, 9-16
-Y compiler option, 9-17

Z

+z compiler option, 9-26
+Z compiler option, 9-26
-z compiler option, 9-17
-Z compiler option, 9-17