HP C

Language Reference Manual

Order Number: AA–PUNDJ–TK

January 2005

This document is the language reference manual for HP C.

Revision Update Information: This manual supersedes the *Compaq C Language Reference Manual* Order No. AA–PUNDH–TK, Version 6.5

Software Version: HP C Version 7.1 for OpenVMS Systems
HP C for Tru64 UNIX Version 5.1 or higher

Hewlett-Packard Company
Palo Alto, California

© Copyright 2005 Hewlett-Packard Development Company, L.P.

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Intel and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group.

X/Open is a registered trademark of X/Open Company Ltd. in the UK and other countries.

Printed in the US

This document is available on CD–ROM.

ZK6180

This document was prepared using DECdocument, Version 3.3-1n.

Contents

Preface xiii

1 Lexicon

1.1 Character Set 1–3
1.1.1 Trigraph Sequences 1–5
1.1.2 Digraph Sequences 1–6
1.2 Identifiers 1–6
1.3 Universal Character Names 1–8
1.4 Comments 1–8
1.5 Keywords 1–9
1.6 Operators 1–11
1.7 Punctuators 1–12
1.8 String Literals 1–13
1.9 Constants 1–14
1.9.1 Integer Constants 1–14
1.9.2 Floating-Point Constants 1–17
1.9.2.1 Semantics 1–18
1.9.2.2 Floating-Point Type 1–18
1.9.2.3 Hexadecimal Floating-Point Constants 1–18
1.9.2.4 Examples 1–19
1.9.3 Character Constants 1–19
1.9.3.1 Wide Characters 1–20
1.9.3.2 Multibyte Characters 1–21
1.9.3.3 Character Escape Sequences 1–21
1.9.3.4 Numeric Escape Sequences 1–22
1.9.4 Enumeration Constants 1–23
1.10 Header Files 1–24
1.11 Limits 1–24
1.11.1 Translation Limits 1–25
1.11.2 Numerical Limits 1–25
1.11.3 Character Display 1–26

2 Basic Concepts

2.1 Blocks 2–2
2.2 Compilation Units 2–3
2.3 Scope 2–3
2.3.1 File Scope 2–4
2.3.2 Block Scope 2–4
2.3.3 Function Scope 2–5
2.3.4 Function Prototype Scope 2–5
2.4 Visibility 2–6
2.5 Side Effects and Sequence Points 2–6
2.6 Incomplete Type 2–8
2.7 Compatible and Composite Types 2–9
2.8 Linkage 2–11
2.9 Tentative Definitions 2–13
2.10 Storage Classes 2–13
2.10.1 The auto Class 2–14
2.10.2 The register Class 2–15
2.10.3 The static Class 2–15
2.10.4 The extern Class 2–15
2.11 Storage-Class Modifiers 2–16
2.11.1 The __inline Modifier 2–17
2.11.2 The inline Modifier 2–17
2.11.2.1 Example—Using the inline Function Specifier 2–19
2.11.3 The __forceinline Modifier 2–21
2.11.4 The __align Modifier 2–21
2.12 Forward References 2–23
2.13 Tags 2–24
2.14 lvalues and rvalues 2–25
2.15 Name Spaces 2–26
2.16 Preprocessing 2–27
2.17 Type Names 2–28

3 Data Types

3.1 Data Sizes 3–4
3.2 Integral Types 3–6
3.2.1 Non-Character Types 3–6
3.2.2 Character Types 3–8
3.3 Floating-Point Types 3–9
3.3.1 Complex Type *(Alpha, I64)* 3–10
3.3.2 Imaginary Type *(Alpha, I64)* 3–11
3.4 Derived Types 3–11

3.4.1 Function Type . . . 3–11
3.4.2 Pointer Type . . . 3–12
3.4.3 Array Type . . . 3–12
3.4.4 Structure Type . . . 3–13
3.4.5 Union Type . . . 3–15
3.5 void Type . . . 3–17
3.6 Enumerated Types . . . 3–18
3.7 Type Qualifiers . . . 3–19
3.7.1 const Type Qualifier . . . 3–20
3.7.2 volatile Type Qualifier . . . 3–22
3.7.3 __unaligned Type Qualifier . . . 3–23
3.7.4 __restrict Type Qualifier . . . 3–24
3.7.4.1 Rationale . . . 3–24
3.7.4.1.1 Aliasing . . . 3–24
3.7.4.1.2 Library Examples . . . 3–25
3.7.4.1.3 Overlapping Objects . . . 3–26
3.7.4.1.4 Restricted Pointer Prototype for memcpy . . . 3–27
3.7.4.2 Formal Definition of the __restrict Type Qualifier . . . 3–27
3.7.4.3 Examples . . . 3–29
3.7.4.3.1 File Scope Restricted Pointers . . . 3–29
3.7.4.3.2 Function Parameters . . . 3–30
3.7.4.3.3 Block Scope . . . 3–30
3.7.4.3.4 Members of Structures . . . 3–31
3.7.4.3.5 Type Definitions . . . 3–31
3.7.4.3.6 Expressions Based on Restricted Pointers . . . 3–32
3.7.4.3.7 Assignments Between Restricted Pointers . . . 3–33
3.7.4.3.8 Assignments to Unrestricted Pointers . . . 3–34
3.7.4.3.9 Ineffective Uses of Type Qualifiers . . . 3–34
3.7.4.3.10 Constraint Violations . . . 3–35
3.8 Type Definition . . . 3–35

4 Declarations

4.1 Declaration Syntax Rules . . . 4–2
4.2 Initialization . . . 4–4
4.3 External Declarations . . . 4–7
4.4 Declaring Simple Objects . . . 4–9
4.4.1 Initializing Simple Objects . . . 4–10
4.4.2 Declaring Integer Objects . . . 4–10
4.4.3 Declaring Character Variables . . . 4–10
4.4.4 Declaring Floating-Point Variables . . . 4–11
4.5 Declaring Enumerations . . . 4–11
4.6 Declaring Pointers . . . 4–12

4.6.1 Declaring void Pointers 4–14
4.6.2 Initializing Pointers 4–15
4.7 Declaring Arrays 4–15
4.7.1 Initializing Arrays 4–18
4.7.2 Pointers and Arrays 4–21
4.7.3 Variable-Length Arrays 4–22
4.8 Declaring Structures and Unions 4–23
4.8.1 Similarities Between Structures and Unions 4–26
4.8.2 Differences Between Structures and Unions 4–27
4.8.3 Bit Fields 4–28
4.8.4 Initializing Structures 4–30
4.8.5 Initializing Unions 4–32
4.9 Initializers with Designations 4–33
4.9.1 Current Object 4–33
4.9.2 Designations 4–33
4.9.3 Examples 4–34
4.10 Declaring Tags 4–36
4.11 Declaring Type Definitions 4–37

5 Functions

5.1 Function Calls 5–1
5.2 Function Types 5–2
5.3 Function Definitions 5–3
5.4 Function Declarations 5–6
5.5 Function Prototypes 5–8
5.5.1 Prototype Syntax 5–9
5.5.2 Scope and Conversions 5–10
5.6 Parameters and Arguments 5–11
5.6.1 Argument Conversions 5–12
5.6.2 Function and Array Identifiers as Arguments 5–12
5.6.3 Passing Arguments to the main Function 5–14

6 Expressions and Operators

6.1 Primary Expressions 6–2
6.1.1 Identifiers 6–2
6.1.2 Constants 6–2
6.1.3 String Literals 6–2
6.1.4 Parenthesized Expressions 6–2
6.2 Overview of the C Operators 6–3
6.3 Postfix Operators 6–7
6.3.1 Array References 6–7

6.3.2 Function Calls 6–8
6.3.3 Structure and Union References 6–10
6.3.4 Postfix Increment and Decrement Operators 6–11
6.4 Unary Operators 6–12
6.4.1 Unary Plus and Minus 6–13
6.4.2 Logical Negation 6–13
6.4.3 Prefix Increment and Decrement Operators 6–13
6.4.4 Address Operator and Indirection 6–15
6.4.5 Bitwise Negation 6–15
6.4.6 The Cast Operator 6–15
6.4.7 The sizeof Operator 6–17
6.4.8 The __typeof__ Operator 6–18
6.4.9 The _Pragma Operator 6–19
6.5 Binary Operators 6–20
6.5.1 Multiplicative Operators 6–20
6.5.2 Additive Operators 6–21
6.5.3 Shift Operators 6–22
6.5.4 Relational Operators 6–23
6.5.5 Equality Operators 6–23
6.5.6 Bitwise Operators 6–24
6.5.7 Logical Operators 6–24
6.6 Conditional Operator 6–25
6.7 Assignment Operators 6–26
6.8 Comma Operator 6–27
6.9 Constant Expressions 6–28
6.9.1 Integral Constant Expressions 6–28
6.9.2 Arithmetic Constant Expressions 6–29
6.9.3 Address Constants 6–29
6.10 Compound Literal Expressions 6–29
6.11 Data-Type Conversions 6–32
6.11.1 Usual Arithmetic Conversions 6–33
6.11.1.1 Characters and Integers 6–34
6.11.1.2 Signed and Unsigned Integers 6–35
6.11.1.3 Floating and Integral 6–35
6.11.1.4 Floating Types 6–36
6.11.2 Pointer Conversions 6–36
6.11.3 Function Argument Conversions 6–37

7 Statements

7.1 Labeled Statements 7–1
7.2 Compound Statements 7–2
7.3 Expression Statements 7–3
7.4 Null Statements 7–3
7.5 Selection Statements 7–4
7.5.1 The if Statement 7–4
7.5.2 The switch Statement 7–5
7.6 Iteration Statements 7–8
7.6.1 The while Statement 7–8
7.6.2 The do Statement 7–9
7.6.3 The for Statement 7–9
7.7 Jump Statements 7–11
7.7.1 The goto Statement 7–11
7.7.2 The continue Statement 7–11
7.7.3 The break Statement 7–12
7.7.4 The return Statement 7–12

8 Preprocessor Directives and Predefined Macros

8.1 Macro Definition (#define and #undef) 8–2
8.1.1 Object-Like Form 8–4
8.1.2 Function-Like Form 8–4
8.1.2.1 Rules for Specifying Macro Definitions 8–5
8.1.2.2 Rules for Specifying Macro References 8–6
8.1.2.3 Side Effects in Macro Arguments 8–6
8.1.3 Conversions to String Literals (#) 8–7
8.1.4 Token Concatenation(##) 8–8
8.2 Conditional Compilation (#if, #ifdef, #ifndef, #else, #elif, #endif, and defined) 8–8
8.2.1 The #if Directive 8–9
8.2.2 The #ifdef Directive 8–9
8.2.3 The #ifndef Directive 8–9
8.2.4 The #else Directive 8–10
8.2.5 The #elif Directive 8–10
8.2.6 The #endif Directive 8–10
8.2.7 The defined Operator 8–11
8.3 File Inclusion (#include) 8–11
8.4 Explicit Line Numbering (#line) 8–12
8.5 Implementation-Specific Preprocessor Directive (#pragma) 8–13
8.6 Error Directive (#error) 8–16
8.7 Null Directive (#) 8–16

8.8 Predefined Macro Names 8–16
8.8.1 The __DATE__ Macro 8–16
8.8.2 The __FILE__ Macro 8–16
8.8.3 The __LINE__ Macro 8–16
8.8.4 The __TIME__ Macro 8–17
8.8.5 The __STDC__, __STDC_HOSTED__, and __STDC_VERSION Macros 8–17
8.8.6 The __STDC_ISO_10646__ Macro 8–17
8.8.7 System-Identification Macros 8–17
8.9 The __func__ Predeclared Identifier 8–18

9 The ANSI C Standard Library

9.1 Diagnostics (<assert.h>) 9–2
9.2 Complex Arithmetic(<complex.h>) 9–2
9.3 Character Processing (<ctype.h>) 9–8
9.4 Error Codes (<errno.h>) 9–10
9.5 ANSI C Limits (<limits.h> and <float.h>) 9–11
9.6 Localization (<locale.h>) 9–11
9.7 Mathematics (<math.h>) 9–16
9.8 Nonlocal Jumps (<setjmp.h>) 9–19
9.9 Signal Handling (<signal.h>) 9–20
9.10 Variable Arguments (<stdarg.h>) 9–21
9.11 Boolean Type and Values(<stdbool.h>) 9–22
9.12 Common Definitions (<stddef.h>) 9–23
9.13 Standard Input/Output (<stdio.h>) 9–24
9.14 General Utilities (<stdlib.h>) 9–35
9.15 String Processing (<string.h>) 9–42
9.16 Type-generic Math (<tgmath.h>) 9–46
9.16.1 Real-Type Determination 9–47
9.16.2 Unsuffixed Functions in (<math.h>) and (<complex.h>) with the Same Name 9–47
9.16.3 Unsuffixed Functions in (<math.h>) with no c-prefixed Counterpart in (<complex.h>) 9–48
9.16.4 Unsuffixed Functions in (<complex.h>) that are not c-prefixed Counterparts to Functions in (<math.h>) 9–48
9.16.5 Example 9–48
9.16.6 Imaginary Arguments 9–49
9.17 Date and Time (<time.h>) 9–50

A Language Syntax Summary

A.1 Lexical Grammar A–1
A.1.1 Tokens A–1
A.1.2 Keywords A–2
A.1.3 Identifiers A–2
A.1.4 Constants A–3
A.1.5 String Literals A–6
A.1.6 Operators A–6
A.1.7 Punctuators A–6
A.1.8 Header Names A–6
A.1.9 Preprocessing Numbers A–7
A.2 Phrase Structure Grammar A–7
A.2.1 Expressions A–7
A.2.2 Declarations A–10
A.2.3 Statements A–13
A.2.4 External Definitions A–14
A.3 Preprocessing Directives A–15

B ANSI Conformance Summary

B.1 Diagnostics (§2.1.1.3) B–2
B.2 Hosted Environment (§2.1.2.2) B–2
B.3 Multibyte Characters (§2.2.1.2) B–2
B.4 Escape Sequences (§2.2.2) B–2
B.5 Translation Limits (§2.2.4.1) B–2
B.6 Numerical Limits (§2.2.4.2) B–3
B.7 Keywords (§3.1.1) B–4
B.8 Identifiers (§3.1.2) B–4
B.9 Linkages of Identifiers (§3.1.2.2) B–5
B.10 Types (§3.1.2.5) B–5
B.11 Integer Constants (§3.1.3.2) B–5
B.12 Character Constants (§3.1.3.4) B–5
B.13 String Literals (§3.1.4) B–5
B.14 Operators—Compound Assignment (§3.1.5) B–6
B.15 Characters and Integers—Value-Preserving Promotions (§3.2.1.1) B–6
B.16 Signed and Unsigned Integer Conversions (§3.2.1.2) B–6
B.17 Floating and Integral Conversions (§3.2.1.3) B–7
B.18 Pointer Conversions (§3.2.2.3) B–7
B.19 Structure and Union Members (§3.3.2.3) B–7
B.20 The sizeof Operator (§3.3.3.4) B–8
B.21 Cast Operators (§3.3.4) B–8

B.22 Multiplicative Operators (§3.3.5) B–8
B.23 Additive Operators (§3.3.6) B–9
B.24 Bitwise Shift Operators (§3.3.7) B–9
B.25 Storage-Class Specifiers (§3.5.1) B–9
B.26 Type Specifiers (§3.5.2) B–9
B.27 Structure and Union Specifiers (§3.5.2.1) B–9
B.28 Variant Structures and Unions B–9
B.29 Structure Alignment B–10
B.30 Enumeration Specifiers (§3.5.2.2) B–11
B.31 Type Qualifiers (§3.5.3) B–11
B.32 Declarators (§3.5.4) B–12
B.33 Initialization (§3.5.7) B–12
B.34 The switch Statement (§3.6.4.2) B–12
B.35 External Object Definitions (§3.7.2) B–12
B.36 Conditional Inclusion (§3.8.1) B–12
B.37 Source File Inclusion (§3.8.2) B–13
B.38 Macro Replacement—Predefined Macro Names (§3.8.3) B–13
B.39 The ## Operator (§3.8.3.3) B–15
B.40 Error Directive (§3.8.5) B–16
B.41 Pragma Directive (§3.8.6) B–16
B.42 Function Inline Expansion B–16
B.43 Linkage Pragmas B–17
B.44 Other Pragmas B–17

C ASCII Equivalence Table

D Common C Extensions Supported by HP C

D.1 Extensions Compatible with ANSI C D–1
D.2 Extensions Incompatible with ANSI C D–3

E VAX C Extensions Supported by HP C

E.1 Extensions Compatible with ANSI C E–1
E.2 Extensions Incompatible with ANSI C E–3

F Universal Character Names for Identifiers

Index

Examples

4–1 The Rules for Initializing Structures 4–30
5–1 Declaring Functions Passed as Arguments 5–13
7–1 Using switch to Count Blanks, Tabs, and New Lines 7–6

Figures

C–1 ASCII Equivalence Chart C–2

Tables

1–1 Trigraph Sequences 1–5
1–2 Digraph Sequences 1–6
1–3 Keywords 1–9
1–4 Punctuators 1–12
1–5 Rules for Assigning Type to Integer Constants 1–16
1–6 Floating-Point Notation 1–19
1–7 Character Escape Sequences 1–22
2–1 Predefined Alignment Constants 2–22
2–2 Type Name Examples 2–28
3–1 Basic Data Types 3–2
3–2 Sizes and Ranges of Data Types 3–4
6–1 C Operators 6–3
6–2 Precedence of C Operators 6–5
9–1 File Modes 9–27
9–2 strftime Conversion Specifiers 9–51
B–1 Tru64 UNIX Predefined Macro Names B–13
B–2 OpenVMS VAX, OpenVMS Alpha, and I64 Predefined Macro Names B–14
B–3 Library Routine Standards Conformance Macros—All platforms B–15

Preface

This manual provides reference information for using the HP C language on HP systems. HP C is an ANSI compliant C compiler for the OpenVMS operating system on VAX, Alpha, and Intel Itanium processors and for the Tru64 UNIX operating system on Alpha processors. HP OpenVMS Industry Standard 64 for Integrity Servers is the full product name of the OpenVMS operating system on Intel Itanium processors. The shortened forms, OpenVMS I64 and I64, are also used throughout this manual.

HP C is compliant with the International Standards Organization (ISO) C Standard (ISO 9899:1990[1992]), formerly the American National Standard for Information Systems-Programming Language C (document number: X3.159-1989). By the use of command-line options, HP C is compatible with older dialects of C, including common usage C (Kernighan and Ritchie C) and VAX C.

This manual is based on the ISO C Standard (ISO 9899:1990[1992]), formerly the ANSI X3J11 committee's standard for the C programming language (called the ANSI C standard in this manual). [1] All library functions and language extensions to the ANSI C standard are also described.

You may send comments or suggestions regarding this manual or any HP C document by sending electronic mail to the following Internet address:

c_docs@hp.com

[1] HP would like to thank CBEMA and its Accredited Standards Committee X3 for use of the material derived in whole or in part from the American National Standard Programming Language C. The ANSI C standard may be purchased from the ANSI Sales Department by calling the United States telephone number 1-212-642-4900.

Intended Audience

This manual is intended for programmers who need reference information on the HP C (formerly Compaq C) language. There is little task-oriented material or platform-specific material in this manual; for that type of information, see your platform-specific HP C documentation (user's guide and online help for OpenVMS systems, programmer's guide and reference pages for Tru64 UNIX systems.)

Purpose of the ANSI Standard

The ANSI C standard was developed by a committee of program developers and knowledgeable C users to address the problems caused by inexact specification of the C language. These problems were primarily related to portability of programs between different types of machines. The committee analyzed the language for areas where its syntax and semantics were vague or indeterminate, and then chose precise definitions for those C constructs. The result is an unambiguous, machine-independent definition.

The ANSI C standard states that it:

'' specifies the form and establishes the interpretation of programs expressed in the programming language C. [The standard's] purpose is to promote portability, reliability, maintainability, and efficient execution of C language programs on a variety of computing systems. ''

The standard specifies:

- Representation, syntax, and constraints of the C language
- Semantic rules for interpreting C programs
- Representation of input and output in C programs

The ANSI C standard does *not* specify:

- How C programs are compiled
- How C programs are linked
- How C programs are executed
- All minimum or maximum limits on the size of machines running ANSI C programs

Manual Structure

This manual has the following chapters and appendixes:

Chapter 1 describes the elements of the C language.

Chapter 2 discusses some of the basic concepts underlying the C language.

Chapter 3 explains HP C data types and type qualifiers.

Chapter 4 describes the declaration of identifiers in HP C. The declaration of constants, variables, structures, unions, pointers, and arrays is covered.

Chapter 5 describes function calls, function declarations, function definitions, function parameters, and function arguments.

Chapter 6 discusses the types of expressions you can build in C. It also explains the effects of operators available in C, including unary, binary, conditional, primary, and postfix operators.

Chapter 7 describes the C statements that provide flow control, conditional executions, looping, and interruption.

Chapter 8 explains the purpose of the C preprocessor directives and predefined macros.

Chapter 9 lists and describes the functions, macros, and types in the ANSI C standard library, arranged by header file.

Appendix A provides a syntax summary of all C language constructs.

Appendix B describes the extent of the ANSI conformance of HP C, including exceptions and extensions to the standard.

Appendix C provides the ASCII octal, decimal, and hexadecimal equivalents for each character in the ASCII character set.

Appendix D lists the common C extensions supported by HP C using the common C compatibility option.

Appendix E lists the VAX C extensions supported by HP C using the VAX C compatibility option.

Associated Documents

You may find the following documents useful when programming in HP C:

- *HP C User's Guide for OpenVMS Systems*—This guide contains the information necessary for developing and debugging HP C programs on the OpenVMS operating system. This guide also includes HP C features specific to OpenVMS systems, as well as information about porting C programs to and from OpenVMS and other operating systems.
- *HP C Run-Time Library Reference Manual for OpenVMS Systems*— Provides complete reference information on the HP C library functions included with the OpenVMS operating system.
- *cc(1) reference page*—This reference page describes the cc command line options for HP C on *Tru64 UNIX* systems.
- *Tru64 UNIX documentation set*—This documentation set provides information about the Tru64 UNIX operating system and its utilities. The following volumes are especially useful:
  - *Tru64 UNIX Programmer's Guide*—This guide describes the Tru64 UNIX programming environment, including information necessary for developing and debugging C programs on the Tru64 UNIX operating system. This guide, together with the cc(1) reference page, includes HP C features specific to Tru64 UNIX systems.
  - Tru64 UNIX Reference Pages, Sections 2 and 3—Provides complete reference information on the C library functions included with the Tru64 UNIX operating system.
- *ANSI/ISO/IEC 9899:1999 - Programming Languages - C*—The C99 standard, published by ISO in December, 1999 and adopted as an ANSI standard in April, 2000.
- *ISO/IEC 9899:1990-1994 - Programming Languages - C, Amendment 1: Integrity*—Documents what is also known as ISO C, Amendment 1.
- *ISO/IEC 9899:1990[1992] - Programming Languages - C*—Documents is also known as ISO C. The normative part is the same as *X3.159-1989, American National Standard for Information Systems - Programming Language C*, also known as ANSI C.
- *American National Standard for Information Systems–Programming Language C*—This document is the result of the X3J11 standards committee analysis of the C language. This document is a very technical description of the ANSI C language, written for knowledgeable C programmers.

- *The C Programming Language, 2nd Edition*[2]—This volume was produced before the final ANSI standard was accepted, but it still serves as a valuable reference to the C language.

  Because ANSI C contains more features and enhancements to the C language than are defined in *The C Programming Language*, use this *HP C Language Reference Manual* as the reference for a full description of HP C.

Conventions Used in This Document

Convention	Meaning
HP OpenVMS Industry Standard 64 for Integrity Servers, OpenVMS I64, I64	The variant of the OpenVMS operating system that runs on the Intel Itanium architecture.
OpenVMS systems	Refers to the OpenVMS operating system on all supported platforms, unless otherwise specified.
Return	The symbol Return represents a single stroke of the Return key on a terminal.
Ctrl/X	The symbol Ctrl/X, where X represents a terminal control character, represents holding down the Ctrl key while pressing the specified terminal character key.
HP C also allows ...	HP C extensions to the ANSI C standard are shown in teal blue in the printed manual and HTML manual.

`float x;`
`.`
`.`
`.`
`x = 5;`
	A vertical ellipsis indicates that not all of the text of a program or program output is shown. Only relevant material is shown in the example.

`option, . . .`	A horizontal ellipsis indicates that additional parameters, options, or values can be entered. A comma preceding the ellipsis indicates that successive items must be separated by commas.

[2] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Englewood Cliffs, New Jersey: Prentice Hall, 1988).

Convention	Meaning
$syntax_{opt}$	Optional syntax elements are indicated with the subscripted abbreviation *opt*. Isolated syntax diagrams in individual sections of this manual may require reference to Appendix A to determine the complete syntax for a construct. For instance, the ANSI C standard syntax includes a constant as a potential *assignment-expression*.
storage-class-specifier : auto static register	In syntax definitions, items appearing on separate lines are mutually exclusive alternatives.
The `auto` storage class . . . The `fprintf` function . . .	Monospaced type identifies language keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.

New and Changed Features

HP C Version 7.1 runs on OpenVMS Alpha and OpenVMS Industry Standard 64 systems. The compiler behaves much the same on both systems, with some differences, primarily in the support for `#pragma linkage`, built-in functions, default floating-point representation, and predefined macros. These differences are noted in the relevant sections of the documentation.

1
Lexicon

C, like any language, uses a standard grammar and character set. The specific elements that comprise this grammar and character set are described in the following sections:

- Character set (Section 1.1)
- Rules for identifiers in C (Section 1.2)
- Use of universal character names (Section 1.3)
- Use of comments in a program (Section 1.4)
- Keywords (Section 1.5)
- Use of C operators (Section 1.6)
- Use of punctuation characters (Section 1.7)
- Use of character strings in a program (Section 1.8)
- Interpretation of constant values (Section 1.9)
- Inclusion of function declarations and other definitions, common to multiple source files, in a separate header file or module (Section 1.10)
- The limits imposed on a conforming program by the ANSI C standard (Section 1.11)

C compilers interpret source code as a stream of characters from the source file. These characters are grouped into *tokens*, which can be punctuators, operators, identifiers, keywords, string literals, or constants. Tokens are the smallest lexical element of the language. The compiler forms the longest token possible from a given string of characters; the token ends when white space is encountered, or when the next character could not possibly be part of the token.

White space can be a space character, new-line character, tab character, form-feed character, or vertical tab character. Comments are also considered white space. Section 1.1 lists all the white space characters. White space is used as a token separator (except within quoted strings), but is otherwise ignored in the character stream, and is used mainly for human readability. White space may also be significant in preprocessor directives (see Chapter 8).

Consider the following source code line:

```
static int x=0;  /* Could also be written "static int x = 0;"   */
```

The compiler breaks the previous line into the following tokens (shown one per line):

```
static
int
x
=
0
;
```

As the compiler processes the input character stream, it identifies tokens and locates error conditions. The compiler can identify three types of errors:

- Lexical errors, which occur when the compiler cannot form a legal token from the character stream (such as when an illegal character is used).
- Parsing (syntax) errors, which occur when a legal token can be formed, but the compiler cannot make a legal statement from the tokens. For example, the following line contains incorrect punctuation surrounding an initializer list:

  ```
  char x[3] = (1,2,3);
  ```

- Semantic errors, which are grammatically correct but break another C language rule. For example, the following line shows an attempt to assign a floating-point value to a pointer type:

  ```
  int *x = 5.7;
  ```

Logical errors are not identified by the compiler.

An important concept throughout C is the idea of a *compilation unit*, which is one or more files compiled by the compiler.

Note

The ANSI C standard refers to compilation units as *translation units*. This text treats these terms as equivalent.

The smallest acceptable compilation unit is one external definition. The ANSI C standard defines several key concepts in terms of compilation units. Section 2.2 discusses compilation units in detail.

A compilation unit with no declarations is accepted with a compiler warning in all modes except for the strict ANSI standard mode.

1.1 Character Set

A *character set* defines the valid characters that can be used in source programs or interpreted when a program is running. The *source character set* is the set of characters available for the source text. The *execution character set* is the set of characters available when executing a program. The source character set does not necessarily match the execution character set; for example, when the execution character set is not available on the devices used to produce the source code.

Different character sets exist; for example, one character set is based on the American Standard Code for Information Interchange (ASCII) definition of characters, while another set includes the Japanese kanji characters. The character set in use makes no difference to the compiler; each character simply has a unique value. C treats each character as a different integer value. The ASCII character set has fewer than 255 characters, and these characters can be represented in 8 bits or less. However, in some extended character sets, so many characters exist that some characters' representation requires more than 8 bits. A special type was created to accommodate these larger characters, called the `wchar_t` (or wide character) type. Section 1.9.3.1 discusses wide characters further.

Most ANSI-compatible C compilers accept the following ASCII characters for both the source and execution character sets. Each ASCII character corresponds to a numeric value. Appendix C lists the ASCII characters and their numeric values.

- The 26 lowercase Roman characters:

  ```
  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
  ```

- The 26 uppercase Roman characters:

  ```
  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
  ```

- The 10 decimal digits:

  ```
  0 1 2 3 4 5 6 7 8 9
  ```

- The 30 graphic characters:

  ```
  ! # % ^ & * ( ) - _ = + ~ ' " : ; ? / | \ { } [ ] , . < > $
  ```

  A warning is issued if the $ character is used when the compiler's strict ANSI mode option is specified.

- Five white space characters:

Space	( )
Horizontal tab	(\t)
Form feed	(\f)
Vertical tab	(\v)
New-line character	(\n)

In character constants and string literals, characters from the execution character set can also be represented by character or numeric escape sequences. Section 1.9.3.3 and Section 1.9.3.4 describe these escape sequences.

The ASCII execution character set also includes the following control characters:

- New-line character (represented by \n in the source file),
- Alert (bell) tone (\a)
- Backspace (\b)
- Carriage return (\r)
- Null character (\0)

The *null character* is a byte or wide character with all bits set to 0. It is used to mark the end of a character string. Section 1.8 discusses character strings in more detail.

The *new-line character* splits the source character stream into separate lines for greater legibility and for proper operation of the preprocessor.

Sometimes a line longer than the terminal or window width must be interpreted by the compiler as one logical line. One logical line can be typed as two or more lines by appending the backslash character (\) to the end of the continued lines. The backslash must be immediately followed by a new-line character. The backslash signifies that the current logical line continues on the next line. For example:

```
#define ERROR_TEXT "Your entry was outside the range of \
0 to 100."
```

The compiler deletes the backslash character and the adjacent new-line character during processing, so that this line becomes one logical line, as follows:

```
#define ERROR_TEXT "Your entry was outside the range of 0 to 100."
```

A long string can be continued across multiple lines by using the backslash-newline line continuation feature, but the continuation of the string must start in the first position of the next line. In some cases, this destroys the indentation scheme of the program. The ANSI C standard introduces another string continuation mechanism to avoid this problem. Two string literals, with only white space separating them, are combined to form one logical string literal. For example:

```
printf ("Your entry was outside the range of "
        "0 to 100.\n");
```

The maximum logical line length is 32,767 characters.

1.1.1 Trigraph Sequences

To write C programs using character sets that do not contain all of C's punctuation characters, ANSI C allows the use of nine *trigraph sequences* in the source file. These three-character sequences are replaced by a single character in the first phase of compilation. (See Section 2.16 for an explanation of compilation phases.) Table 1–1 lists the valid trigraph sequences and their character equivalents.

Table 1–1 Trigraph Sequences

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

No other trigraph sequences are recognized. A question mark (?) that does not begin a trigraph sequence remains unchanged during compilation. For example, consider the following source line:

```
printf ("Any questions???/n");
```

After the ??/ sequence is replaced, this line is translated as follows:

```
printf ("Any questions?\n");
```

1.1.2 Digraph Sequences

Digraph processing is supported when compiling in ISO C 94 mode (/STANDARD=ISOC94 on OpenVMS systems).

Digraphs are pairs of characters that translate into a single character, much like trigraphs, except that trigraphs get replaced inside string literals, but digraphs do not. Table 1–2 lists the valid digraph sequences and their character equivalents.

Table 1–2 Digraph Sequences

Digraph Sequence	Character Represented
<:	[
:>	]
<%	{
%>	}
%:	#
%:%:	##

1.2 Identifiers

An *identifier* is a sequence of characters that represents a name for the following:

- Variable
- Function
- Label
- Type definition
- Structure, enumeration, or union tag
- Structure, enumeration, or union member

- Enumeration constant
- Macro
- Macro parameter

The following rules apply to identifiers:

- Identifiers consist of a sequence of one or more: uppercase or lowercase alphabetic characters, universal character names, the digits 0 to 9, the dollar sign ($), and the underscore character (_).

  Using the $ character provokes a warning from the compiler in strict ANSI mode.

- Character case is significant in identifiers; for example, the identifier `Test1` is different from the identifier `test1`.
- Identifiers cannot begin with a digit.
- Do not begin identifiers with an underscore; the ANSI C standard reserves such identifiers for internal names.
- Each universal character name in an identifier must designate a character whose encoding in ISO/IEC 10646 falls into one of the ranges specified in Appendix F.
- Keywords are not identifiers (Section 1.5 lists the C keywords).
- Using the names of library functions for identifiers is bad practice (Chapter 9 lists the C library function names). A function with the same name as a library function will supersede the library function. This may be the desired outcome, but program maintenance can be confusing.
- In general, identifiers are separated by white space, punctuators, or operators. For example, the following code fragment has four identifiers:

  ```
  struct employee { int number; char sex; } emp;
  ```

  The identifiers are: `employee`, `number`, `sex`, and `emp`. (`struct`, `int`, and `char` are keywords).

An identifier *without* external linkage has at most 32,767 significant characters. An identifier *with* external linkage has 1023 significant characters on Tru64 UNIX systems and 31 significant characters for OpenVMS platforms. (Section 2.8 describes linkage in more detail.) Case is not significant in external identifiers on OpenVMS systems.

Identifiers that differ within their significant characters are different identifiers. If two or more identifiers differ in nonsignificant characters only, they are treated as the same identifier.

1.3 Universal Character Names

Universal character names provide a way to name other characters. They can be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set.

A universal character name begins with a **\u** or **\U** and is followed by either four or eight hexadecimal digits.

The universal character name \Unnnnnnnn designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is nnnnnnnn) Similarly, the universal character name \unnnn designates the character whose four-digit short identifier is nnnn (and whose eight-digit short identifier is 0000nnnn).

A universal character name cannot specify a character whose short identifier is less than 00A0, other than 0024 ($), 0040 (@), or 0060 (‘), nor one in the range D800 through DFFF inclusive.)

See Appendix F for a list of valid universal character names.

1.4 Comments

Except within a character constant, string literal, or a comment, the `/*` character combination introduces a comment and the `*/` character combination ends a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` to terminate it.

Alternatively, the `//` character combination introduces a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

Comments cannot be nested; once a comment is started, the compiler treats the first occurrence of `*/` as the end of the comment.

To comment out sections of code, avoid using the `/*` and `*/` sequences. Using the `/*` and `*/` sequences works only for code sections containing no comments, because comments do not nest. A better method is to use the `#if` and `#endif` preprocessor directives, as in the following example:

```
#if 0
/*  This code is excluded from execution because ...  */
code_to_be_excluded ();
#endif
```

See Chapter 8 for more information on the preprocessing directives `#if` and `#endif`.

Comments cannot span source files. Within a source file, comments can be of any length and are interpreted as white space by both the compiler and the preprocessor.

Examples:

```
"a//b"                    // four-character string literal
#include "//e"           // undefined behavior
// */                     // comment, not syntax error
f = g/**//h;              // equivalent to f = g / h;
//\
i();                      // part of a two-line comment
/\
/ j();                    // part of a two-line comment
#define glue(x,y) x##y
glue(/,/) k();            // syntax error, not comment
/*//*/ l();               // equivalent to l();
m = n//**/o
+ p;                      // equivalent to m = n + p;
```

1.5 Keywords

C defines several *keywords*, each with special meaning to the compiler. Keywords identify statement constructs and specify basic types and storage classes. Keywords cannot be used as identifiers and cannot be declared.

Table 1–3 lists the C keywords.

Table 1–3 Keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while
_Bool	_Complex *(Alpha, I64)*	inline	restrict
_Imaginary *(Alpha, I64)*			

In addition to the keywords listed in Table 1–3, the compiler reserves all identifiers that begin with two underscores (_ _) or with an underscore followed

by an uppercase letter. User variable names must never begin with one of these sequences.

Keywords are used as follows:

- To assign a storage class to a variable or function (`auto`, `extern`, `register`, `static`)
- To construct or qualify a data type (`_Bool`, `char`, `_Complex` *(Alpha, I64)*, `const`, `double`, `enum`, `float`, `int`, `long`, `short`, `signed`, `struct`, `union`, `unsigned`, `void`, `volatile`)
- As part of a statement (`break`, `case`, `continue`, `default`, `do`, `else`, `for`, `goto`, `if`, `return`, `switch`, `while`)
- To define a new named type (`typedef`)
- To perform an operation (`sizeof`, `__typeof__`)

The following VAX C keywords are also sometimes[1] recognized by the compiler:

```
_align
globaldef
globalref
globalvalue
noshare
readonly
variant_struct
variant_union
```

The following C99 Standard keywords are also sometimes[2] recognized by the compiler:

```
inline
restrict
```

Use of a keyword as a superfluous macro name is not recommended, but is legal; for example, to change the default size of a basic data type:

```
#define int short
```

Here, the keyword `int` has been redefined as `short`, which causes all data objects declared with the `int` data type to be stored as `short` objects.

---

[1] Recognized on OpenVMS systems when /STANDARD=RELAXED (the default), /STANDARD=VAXC or /ACCEPT=VAXC_KEYWORDS is specified on the compiler command line. Recognized on Tru64 UNIX systems when `-vaxc` or `-accept vaxc_keywords` is specified on the compiler command line.

[2] Recognized on OpenVMS systems when /STANDARD=RELAXED (the default), /STANDARD=C99, or /ACCEPT=C99_KEYWORDS is specified on the compiler command line. Recognized on Tru64 UNIX systems when `-std` (the default), -c99, or `-accept c99_keywords` is specified on the compiler command line.

1.6 Operators

An *operator* is a token that specifies an operation on at least one *operand*, and yields some result (a value, designator, side effect, or some combination). Operands are expressions or constants (a form of expression). Operators with one operand are *unary operators*, and operators with two operands are *binary operators*. For example:

```
x = -b;          /*   Unary minus operator   */
y = a - c;       /*   Binary minus operator  */
```

Operators with three operands are called *ternary operators*.

All operators are ranked by *precedence*, a ranking system determining which operators are evaluated before others in a statement. See Chapter 6 for information on what each operator does and for the rules of operator precedence.

Some operators in C are composed of more than one character, while others are single characters. The single-character operators in C are:

```
!  %  ^  &  *  -  +  =  ~  |  .  <  >  /  ?  :  ,  [  ]  (  )  #
```

The multiple-character operators in C are:

```
++    --    ->    <<    >>    <=    >=    ==    !=    *=    /=
%=    +=    -=    <<=   >>=   &=    ^=    |=    ##    &&    ||
```

The # and ## operators can only be used in preprocessor macro definitions. See Chapter 8 for more information on predefined macros and preprocessor directives.

The `sizeof` operator determines the size of a data type. See Chapter 6 for more information on the `sizeof` operator.

The old form for compound assignment operators (=+, =-, =*, =/, =%, =<<, =>>, =&, =^, and =|) is not supported by the ANSI C standard. Use of these operators in a program is unsupported, and will produce unpredictable results. For example:

```
x =-3;
```

This construction means `x` is assigned the value `-3`, not `x` is assigned the value `x - 3`.

The error-checking compiler option provides a warning message when the old form of compound assignment operators is encountered.

1.7 Punctuators

Some characters in C are used as *punctuators*, which have their own syntactic and semantic significance. Punctuators are not operators or identifiers. Table 1–4 lists the C punctuators.

Table 1–4 Punctuators

Punctuator	Use	Example
< >	Header name	`<limits.h>`
[ ]	Array delimiter	`char a[7];`
{ }	Initializer list, function body, or compound statement delimiter	`char x[4] = {'H', 'i', '!', '\0' };`
( )	Function parameter list delimiter; also used in expression grouping	`int f (x,y)`
*	Pointer declaration	`int *x;`
,	Argument list separator	`char x[4] = { 'H', 'i', '!', '\0'};`
:	Statement label	`labela: if (x == 0) x += 1;`
=	Declaration initializer	`char x[4] = { "Hi!" };`
;	Statement end	`x += 1;`
...	Variable-length argument list	`int f ( int y, ...)`
#	Preprocessor directive	`#include <limits.h>`
' '	Character constant	`char x = 'x';`
" "	String literal or header name	`char x[ ] = "Hi!";`

The following punctuators must be used in pairs:

```
< >
[ ]
( )
' '
" "
{ }
```

Some characters can be used either as a punctuator or as an operator, or as part of an operator. The context of the occurrence specifies the meaning. Punctuators usually delineate a specific type of C construct, as shown in Table 1–4.

1.8 String Literals

Strings are sequences of zero or more characters. A *character string literal* is a sequence of zero or more multibyte characters enclosed in double quotation marks, as in "xyz". String literals can include any valid character, including white-space characters and character escape sequences. A *wide string literal* is the same, except prefixed by the letter **L**. Once a string is stored as a string literal, modification of the string leads to undefined results.

In the following example, `ABC` is the string literal. It is assigned to a character array where each character in the string literal is stored as one array element. Storing a string literal in a character array lets you modify the characters of the array.

```
char x[] = "ABC";
```

String literals are typically stored as arrays of type `char` (or `wchar_t` if prefaced with an `L`), and have static storage duration.

The following declaration declares a character array to hold the string "Hello!":

```
char s[] = "Hello!";
```

The character array `s` is initialized with the characters specified in the double quotation marks, and terminated with a null character (`\0`). The null character marks the end of each string, and is automatically concatenated to the end of the string literal by the compiler. Adjacent string literals are automatically concatenated (with a single null character added at the end) to reduce the need for the line continuation character (the backslash at the end of a line).

Normal string literals and wide string literals can be concatenated, in which case the normal strings get promoted to wide strings, and a wide-string result is produced.

Following are some valid string literals:

```
""            /*  Here's a string with only the null character */

"You can have many characters in a string."

"\"You can mix characters and escape sequences.\"\n"

"Long lines of text can be continued on the next line \
by using the backslash character at the end of a line."
```

```
"Or, long lines of text can be continued by using "
"ANSI's concatenation of adjacent string literals."

"\'\n"        /*  Only escape sequences are in this string    */
```

To determine the length of a given string literal (not including the null character), use the `strlen` function. See Chapter 9 for more information on other library routines available for string manipulation.

1.9 Constants

There are four categories of constants in C:

- Integer constants (such as `63`, `0`, and `42L`)
- Floating-point constants (such as `1.2`, `0.00`, and `77E+2`)
- Hexadecimal floating-point constants (such as `0x1P-1` or `0x.1P3` to represent 1/2).
- Character constants (such as `'A'`, `'0'`, and `L'\n'`)
- Enumeration constants (such as `enum boolean { NO, YES };`), where `NO` and `YES` are the enumeration constants

The following sections describe these constants.

The value of any constant must be within the range of representable values for the specified type. Regardless of its type, a constant is a literal or symbolic value that does not change. A constant is also an rvalue, as defined in Section 2.14.

1.9.1 Integer Constants

Integer constants are used to represent whole numbers. An integer constant can be specified in decimal, octal, or hexadecimal radix, and can optionally include a prefix that specifies its radix and a suffix that specifies its type. An integer constant cannot include a period or an exponent part.

Follow these rules when specifying an integer constant:

- To specify a decimal integer constant, use a sequence of decimal digits in which the first digit is not 0. The value of a decimal constant is computed in base 10.
- To specify an octal integer constant, start the sequence with a zero (0) and follow the 0 (if necessary) with a sequence composed of the digits 0 to 7. A leading 0 alone signifies the octal number 0. The value of an octal constant is computed in base 8.

- To specify a hexadecimal integer constant, start the hexadecimal sequence with a 0 followed by the character `X` (or `x`). Follow the `X` or `x` with one or more hexadecimal characters (the digits 0 to 9 and the upper or lowercase letters A to F). The value of a hexadecimal constant is computed in base 16 (the letters A to F have the values 10 to 15, respectively).

Without explicit specification, the type of an integer constant defaults to the smallest possible type that can hold the constant's value, unless the value is suffixed with an `L`, `l`, `LL` (*(Alpha, I64)*), `ll` (*(Alpha, I64)*), `U`, or `u`.

The C99 standard introduced the type `long long int` (both signed and unsigned) as a standard integer type whose range of values requires at least 64 bits to represent. Although HP C on Alpha systems implemented the type `long long` as a language extension many releases ago, the compiler followed the C89 rules for determining the type of an integer constant. Those rules specified that an unsuffixed decimal integer with a value too large to be represented in a `signed long` would be given the type `unsigned long` if it would fit, and only be given a `long long` type if the value was too large for `unsigned long`. (Note: The `long long` data type is supported on Alpha and I64 systems only.)

In standardizing the `long long` type, the C99 standard regularized these rules and made them extensible to longer types. In particular, unsuffixed decimal integer constants are given the smallest signed integer type that will hold the value (the minimum type is still `int`). If the value is larger than the largest value of `signed long long`, it is given the next larger implementation-defined signed integer type (if there is one). Otherwise C99 states that the behavior is undefined. HP C, however, uses the type `unsigned long long` next. The only portable way to specify a decimal constant that will be given an unsigned type is to use a suffix containing u or U .

HP C continues to use the C89 rules in VAXC, COMMON, and strict ANSI89 modes (including MIA), but uses the new C99 rules in all other modes. Table 1–5 shows the rules for determining the type of an integer constant. The type of an integer constant is the first of the corresponding list in which its value can be represented.

Table 1–5 Rules for Assigning Type to Integer Constants

Suffix	Decimal Constant	Octal or Hexadecimal Constant
none	`int`	`int`
	`long int`	`unsigned int`
	`long long int`[1]	`long int`
	`unsigned long long int`	`unsigned long int`
		`long long int`
		`unsigned long long int`
u or U	`unsigned int`	`unsigned int`
	`unsigned long int`	`unsigned long int`
	`unsigned long long int`	`unsigned long long int`
l or L	`long int`	`long int`
	`long long int`[1]	`unsigned long int`
	`unsigned long long int`	`long long int`
		`unsigned long long int`
Both u or U and l or L	`unsigned long int`	`unsigned long int`
	`unsigned long long int`	`unsigned long long int`
ll or LL	`long long int`	`long long int`
	`unsigned long long int`	`unsigned long long int`
Both u or U and ll or LL	`unsigned long long int`	`unsigned long long int`

[1]In VAXC, COMMON, ANSI89, and MIA modes, the type `unsigned long int` is assigned before `long long int`.

For example, the constant `59` is assigned the `int` data type; the constant `59L` is assigned the `long` data type; the constant `59UL` is assigned the `unsigned long int` data type.

Integer constant values are always nonnegative; a preceding minus sign is interpreted as a unary operator, not as part of the constant. If the value exceeds the largest representable integer value (causing an overflow), the compiler issues a warning message and uses the greatest representable value for the integer type. Unsuffixed integer constants can have different types, because without explicit specification the constant is represented in the smallest possible integer type.

Note

The new C99 rules for determining the type of an integer constant could lead to some constants in your program being interpreted as having a signed type when previous compiler versions gave them an unsigned type. This could affect your program's behavior in subtle ways. The new message `intconstsigned` can be enabled to report constants in your source code that are being treated differently under the C99 rules than they were in previous releases. This message is also part of the new message group NEWC99. If your program relied on unsigned treatment, the simple fix is to add the correct suffix including a "U" or "u" to force the constant to have the expected type. Such a change would be backward compatible and portable.

1.9.2 Floating-Point Constants

A *floating-point constant* has a *significand part* that may be followed by an *exponential part* and an optional suffix that specifies its type (for example, `32.45E2`).

The components of the significand part may include a digit sequence representing the whole number part, followed by a period (`.`), followed by a digit sequence representing the fractional part.

The components of the exponent part are an e, E, p, or P followed by an exponent consisting of an optionally signed digit sequence.

Either the whole-number part or the fraction part of the significand must be present. For decimal floating constants, either the period or the exponent part must be present.

1.9.2.1 Semantics

The significand part of a floating-point constant is interpreted as a decimal or hexadecimal rational number; the digit sequence in the exponent is interpreted as a decimal integer. For decimal floating constants, the exponent indicates the power of 10 by which the significand part is to be scaled. For hexadecimal floating constants, the exponent indicates the power of 2 by which the significand part is to be scaled. For decimal floating constants, and for hexadecimal floating constants when FLT_RADIX is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an platform-dependent manner. For hexadecimal floating constants when FLT_RADIX is a power of 2, the result is correctly rounded.

Floating-point constant values must be nonnegative; a preceding minus sign is interpreted as a unary operator, not as part of the constant.

1.9.2.2 Floating-Point Type

Floating-point constants have the following type:

- An unsuffixed floating-point constant has type `double`.

  If the value exceeds the largest value representable by type `double`, a compiler overflow warning results. (The result is truncated within the `double` type.)

- If suffixed by the letter f or F, it has type `float`.
- If suffixed by the letter l or L, it has type `long double`.

1.9.2.3 Hexadecimal Floating-Point Constants

A word about hexadecimal floating-point constants... The C99 standard introduced a hexadecimal form of floating-point constants. This form of constant permits floating-point values to be specified reliably to the last bit of precision. It does not specify a bit pattern for the representation. Instead it is interpreted much like an ordinary decimal floating-point constant except that the significand is written in hexadecimal radix, and the exponent is expressed as a decimal integer indicating the power of two by which to multiply the significand. A "P" instead of an "E" separates the exponent from the significand. Thus, for example, 1/2 can be written as either 0x1P-1 or 0x.1P3.

The C99 standard also adds `printf`/`scanf` specifiers for this form of value, but that support will be in OpenVMS run-time libraries after OpenVMS Version 7.3.

1.9.2.4 Examples

Table 1–6 shows examples of valid notational options.

Table 1–6 Floating-Point Notation

Notation	Value	Type
.0	0.000000	`double`
0.	0.000000	`double`
2.	2.000000	`double`
2.5	2.500000	`double`
2e1	20.00000	`double`
2E1	20.00000	`double`
0x1P-1	0.500000	`double`
0x.1P3	0.500000	`double`
2.E+1	20.00000	`double`
2e+1	20.00000	`double`
2e-1	0.200000	`double`
2.5e4	25000.00	`double`
2.5E+4	25000.00	`double`
2.5F	2.500000	`float`
2.5L	2.500000	`long double`

1.9.3 Character Constants

A *character constant* is any character from the source character set enclosed in apostrophes. Character constants are represented by objects of type `int`. For example:

```
char alpha = 'A';
```

Characters such as the new-line character, single quotation marks, double quotation marks, and backslash can be included in a character constant by using escape sequences as described in Section 1.9.3.3. All valid characters can also be included in a constant by using numeric escape sequences, as described in Section 1.9.3.4.

The value of a character constant containing a single character is the numeric value of the character in the current character set. Character constants containing multiple characters within the single quotation marks have a value determined by the compiler. The value of a character constant represented

by an octal or hexadecimal escape sequence is the same as the octal or hexadecimal value of the escape sequence. The value of a wide character constant (discussed in Section 1.9.3.1) is determined by the `mbtowc` library function.

There is a limit of four characters for any one character constant. Enclosing more than four characters in single quotation marks (such as `'ABCDE'`), generates an overflow warning.

Note that the byte ordering of character constants is platform specific.

1.9.3.1 Wide Characters

C provides for an extended character set through the use of *wide characters*. Wide characters are characters too large to fit in the `char` type. The `wchar_t` type is typically used to represent a character constant in a character set requiring more than 256 possible characters, because 8 bits can represent only 256 different values.

A character constant in the extended character set is written using a preceding `L`, and is called a *wide-character constant*. Wide-character constants have an integer type, `wchar_t`, defined in the `<stddef.h>` header file. Wide-character constants can be represented with octal or hexadecimal character escape sequences, just like normal character escape sequences, but with the preceding `L`.

Strings composed of wide characters can also be formed. The compiler allocates storage as if the string were an array of type `wchar_t`, and appends a wide null character (\0) to the end of the string. The array is just long enough to hold the characters in the string and the wide null character, and is initialized with the specified characters.

The following examples show valid wide-character constants and string literals:

```
wchar_t wc = L'A';
wchar_t wmc = L'ABCD';
wchar_t *wstring = L"Hello!";
wchar_t *x = L"Wide";
wchar_t z[] = L"wide string";
```

HP C stores `wchar_t` objects as `unsigned long` objects *(OpenVMS)* or `unsigned int` objects *(Tru64 UNIX)* in 32 bits of storage. The null character at the end of a wide-character string is 32 bits long.

1.9.3.2 Multibyte Characters

Some programmers requiring an extended character set have used shift-dependent encoding schemes to represent the non-ASCII characters in the normal `char` size of 8 bits. This encoding results in *multibyte characters*. ANSI C supports these encoding schemes, in addition to providing the wide-character type `wchar_t`.

In accordance with the ANSI standard, HP C recognizes multibyte characters in the following contexts:

- Comments
- String literals
- Header names
- Character constants

For proper input and output of the multibyte character encodings, and to prevent conflicts with existing string processing routines, note the following rules governing the use of multibyte characters:

- A byte with all bits set to zero is always recognized as a null character. Null characters can only be single bytes.
- A null character cannot occur as the second or subsequent byte of a multibyte character.

Transforming multibyte characters to wide-character constants and wide string literals eases the programmer's problems when dealing with shift-state encoding. There are several C library functions available for transforming multibyte characters to wide characters and back. See Chapter 9 for more information.

1.9.3.3 Character Escape Sequences

Characters that cannot be displayed on a standard terminal, or that have special meaning when used in character constants or string literals, can be entered as source characters by entering them as *character escape sequences*. A backslash (\) begins each character escape sequence. Each of the escape sequences is stored in a single `char` or `wchar_t` object. Table 1–7 lists the ANSI-defined escape sequences.

Table 1–7 Character Escape Sequences

Character	Escape Sequence
Alert (Bell)	\a
Backspace	\b
Form Feed	\f
New line	\n
Carriage Return	\r
Horizontal Tab	\t
Vertical Tab	\v
Backslash	\\
Single Quote	\’
Double Quote	\"
Question Mark	\?

No other character escape sequences are valid. If another sequence is encountered in the source code, the compiler issues a warning and the backslash character is ignored.

An example of a character escape sequence use follows:

```
printf ("\t\aReady\?\n");
```

Upon execution, this results in an alert bell and the following prompt:

```
    Ready?
```

1.9.3.4 Numeric Escape Sequences

The compiler treats all characters as an integer representation, so it is possible to represent any character in the source code with its numeric equivalent. This is called a *numeric escape sequence*. The character is represented by typing a backslash (\), followed by the character's octal or hexadecimal integer equivalent from the current character set (see Appendix C for the ASCII equivalence tables). For example, using the ASCII character set, the character `A` can be represented as `\101` (the octal equivalent) or `\x41` (the hexadecimal equivalent). A preceding `0` in the octal example is not necessary because octal values are the default in numeric escape sequences. A lowercase `x` following the backslash indicates a hexadecimal representation. For example, `\x5A` is equivalent to the character `Z`.

An example of numeric escape sequences follows:

```
#define NUL '\0'   /*  Defines logical null character   */

char x[] = {'\110','\145','\154','\154','\157','\41','\0'};
                    /*  Initializes x with "Hello!"  */
```

The escape sequence extends to three octal digits, or the first character that is not an octal digit, whichever is first. Therefore, the string `"\089"` is interpreted as four characters: `\0`, `8`, `9`, and `\0`.

With hexadecimal escape sequences, there is no limit to the number of characters in the escape sequence, but the result is not defined if the hexadecimal value exceeds the largest value representable by the `unsigned char` type for an normal character constant, or the largest value representable by the `wchar_t` type for a wide-character constant. For example, `'\x777'` is illegal.

In addition, hexadecimal escape sequences with more than three characters provoke a warning if the error-checking compiler option is used.

String concatenation can be used to specify a hexadecimal digit following a hexadecimal escape sequence. In the following example, `a` is initialized to the same value in both cases:

```
char a[] = "\xff" "f";
char a[] = {'\xff', 'f', '\0'};
```

Using numeric escape sequences can result in a nonportable program if the executing machine uses a different character set. Another threat to portability exists if arithmetic operations are performed on the integer character values, because multiple character constants (such as `'ABC'` can be represented differently on different machines.

1.9.4 Enumeration Constants

An enumerated type specifies one or more *enumeration constants* to define allowable values for the enumerated type. Enumeration constants have the type `signed int`, except in the compiler's RELAXED mode, in which other types are allowed. See Section 3.6 for details on the declaration and use of enumerated types.

1.10 Header Files

Header files are text files included in a source file during compilation. To include a header file in a compilation, the `#include` preprocessor directive must be used in the source file. See Chapter 8 for more information on this directive. The entire header file, regardless of content, is substituted for the `#include` preprocessor directive.

A header file can contain other `#include` preprocessor directives to include another file. You can nest `#include` directives to any depth.

Header files can include any legal C source code. They are most often used to include external variable declarations, macro definitions, type definitions, and function declarations. Groups of logically related functions are commonly declared together in a header file, such as the C library input and output functions listed in the `stdio.h` header file. Header files traditionally have a `.h` suffix (`stdio.h`, for example).

The names of header files must not include the ', \, ", or /* characters, because the use of these punctuation characters in a header file is undefined.

When referenced in a program, header names are surrounded by angle brackets or double quotation marks, as shown in the following example:

```
#include <math.h>   /* or  */
#include "local.h"
```

Chapter 8 explains the difference between the two formats. The algorithm the compiler uses for finding the named files is discussed in Section B.37. Chapter 9 describes the library routines in each of the ANSI standard header files.

1.11 Limits

The ANSI C standard suggests several environmental limits on the use of the C language. These limits are an effort to define minimal standards for a conforming implementation of a C compiler. For example, the number of significant characters in an identifier is implementation-defined, with a minimum set required by the ANSI C standard.

The standard also includes several numerical limits that restrict the characteristics of integral and floating-point types. For the most part, these limits will not affect your use of the C language or compiler. However, for unusually large or unusually constructed programs, certain limits can be reached. The ANSI standard contains a list of minimum limits, and your platform-specific HP C documentation contains the actual limits used in HP C.

1.11.1 Translation Limits

As intended by the ANSI C standard, the HP C implementation avoids imposing many of the translation limits, allowing applications more flexibility. The HP C limits are:

- A maximum of 32,767 significant characters in an internal identifier or macro name (a warning message is issued if this limit is exceeded)
- A maximum of 1023 significant characters in an external identifier for Tru64 UNIX systems.
- A maximum of 31 significant characters in an external identifier for OpenVMS VAX platforms (a warning message is issued if this limit is exceeded and the identifier is truncated)
- A maximum of 253 function arguments/formal parameters on OpenVMS systems; a maximum of 1023 function arguments/formal parameters on Tru64 UNIX systems.
- A maximum of 1012 bytes in any one function argument, and a maximum of 1012 bytes in a function argument list on OpenVMS systems
- A maximum of 32,767 characters in a logical source line
- A maximum of 32,767 characters in a physical source line
- A maximum of 32,767 bytes in the representation of a string literal (this limit does not apply to string literals formed as a result of concatenation)

1.11.2 Numerical Limits

Numerical limits define the sizes and characteristics of integral and floating-point types. Numerical limits are described in the `limits.h` and `float.h` header files. The limits are:

- Each character of type `char` is represented in 8 bits.
- Each character of type `wchar_t` is represented in 32 bits.
- The machine representation and set of possible values for the `char` type is the same as for the `signed char` type. A compiler command-line option changes this equivalence to `unsigned char`.
- On OpenVMS systems, the machine representation and set of possible values for the `int` and `signed int` types are the same as for the `long int` type.
- On OpenVMS systems, the machine representation and set of possible values for the `unsigned int` type are the same as for the `unsigned long int` type.

- On Tru64 UNIX systems, the `long int` and `unsigned long int` types are 64 bits, while `int` and `unsigned int` are 32 bits.
- The machine representation and set of possible values for the `long double` type is the same as for the `double` type.

1.11.3 Character Display

Characters from the executable character set are output to the *active position* on the screen or in a file. The active position is defined by the ANSI C standard as the spot where the next output character will appear. After a character is output, the active position advances to the next position on the current line (to the left or right).

The HP C compiler moves the active position from left to right across an output line.

2
Basic Concepts

The C language was initially designed as a small, portable programming language used to implement an operating system. In its history, C has evolved into a powerful tool for writing all types of programs, and includes mechanisms to achieve most programming goals. C offers:

- A standard set of lexical elements
- A wide variety of types for data objects, including:
  - Integer and floating-point constants and variables
  - Pointers to data locations in memory and the ability to do pointer arithmetic
  - Arrays of identically typed data
  - Structures and unions with members of different data types
- The ability to group independent code blocks into named functions
- A large set of operators used to form expressions, including bit-wise operators
- A simple method of declaring data objects and functions
- Several preprocessor directives to expand the functionality of the language
- Numerous library functions to handle many common programming tasks
- A high degree of portability

To help you take full advantage of C's features, the following sections provide a guide to the basic concepts of the language:

- Blocks (Section 2.1)
- Compilation units (Section 2.2)
- Scope (Section 2.3)
- Visibility (Section 2.4)

- Side effects and sequence points (Section 2.5)
- Incomplete type (Section 2.6)
- Compatible and composite types (Section 2.7)
- Linkage (Section 2.8)
- Storage classes (Section 2.10)
- Storage-class modifiers (Section 2.11)
- Forward references (Section 2.12)
- Tags (Section 2.13)
- lvalues and rvalues (Section 2.14)
- Name spaces (Section 2.15)
- Preprocessing (Section 2.16)
- Type names (Section 2.17)

These sections represent an expanded glossary of selected C terms and basic concepts. Understanding these concepts will provide a good foundation for a working knowledge of C, and will help show the relationship of these concepts to more complex ones in the language.

2.1 Blocks

A *block* in C is a section of code surrounded by braces { }. Understanding the definition of a block is very important to understanding many other C concepts, such as scope, visibility, and external or internal declarations.

The following example shows two blocks, one defined inside the other:

```
main ()
{              /*  This brace marks the beginning of the outer block  */
   int x;
   if (x!=0)
   {           /*  This brace marks the beginning of the inner block */
      x = x++;
      return x;
   };          /*  This brace marks the end of the inner block        */
}              /*  This brace marks the end of the outer block        */
```

A block is also a form of a *compound statement*; a set of related C statements enclosed in braces. Declarations of objects used in the program can appear anywhere within a block and affect the object's scope and visibility. Section 2.3 discusses scope; Section 2.4 discusses visibility.

2.2 Compilation Units

A *compilation unit* is C source code that is compiled and treated as one logical unit. The compilation unit is usually one or more entire files, but can also be a selected portion of a file if, for example, the `#ifdef` preprocessor directive is used to select specific code sections. Declarations and definitions within a compilation unit determine the scope of functions and data objects.

Files included by using the `#include` preprocessor directive become part of the compilation unit. Source lines skipped because of the conditional inclusion preprocessor directives are not included in the compilation unit.

Compilation units are important in determining the scope of identifiers, and in determining the linkage of identifiers to other internal and external identifiers. Section 2.3 discusses scope. Section 2.8 discusses linkage.

A compilation unit can refer to data or functions in other compilation units in the following ways:

- A function in one compilation unit can call a function in a different compilation unit.
- Data objects can be assigned external linkage so that other compilation units have access to them (see Section 2.8).

Programs composed of more than one compilation unit can be separately compiled, and later linked to produce the executable program. A legal C compilation unit consists of at least one external declaration, as defined in Section 4.3.

A translation unit with no declarations is accepted with a compiler warning in all modes except for the strict ANSI standard mode.

2.3 Scope

The *scope* of an identifier is the range of the program in which the declared identifier has meaning. An identifier has meaning if it is recognized by the compiler. Scope is determined by the location of the identifier's declaration. Trying to access an identifier outside of its scope results in an error. Every declaration has one of four kinds of scope:

- File
- Block
- Function
- Function prototype (a declaration including only the function's parameter types)

An enumeration constant's scope begins at the defining enumerator in an enumerator list. The scope of a statement label includes the entire function body. The scope of any other type of identifier begins at the identifier itself in the identifier's declaration. See the following sections for information on when an identifier's scope ends.

2.3.1 File Scope

An identifier whose declaration is located outside any block or function parameter list has *file scope*. An identifier with file scope is visible from the declaration of the identifier to the end of the compilation unit, unless hidden by an inner block declaration. In the following example, the identifier `off` has file scope:

```
int off = 5;     /*  Declares (and defines) the integer
                       identifier off.                        */
main ()
{
   int on;       /*  Declares the integer identifier on.     */
   on = off + 1; /*  Uses off, declared outside the function
                     block of main.  This point of the
                     program is still within the
                     active scope of off.                    */
   if (on<=100)
   {
     int off = 0;/*  This declaration of off creates a new object
                     that hides the former object of the same name.
                     The scope of the new off lasts through the
                     end of the if block.                    */
     off = off + on;
     return off;
   }
}
```

2.3.2 Block Scope

An identifier appearing within a block or in a parameter list of a function definition has *block scope* and is visible within the block, unless hidden by an inner block declaration.

Block scope begins at the identifier declaration and ends at the closing brace (}) completing the block. In the following example, the identifier `red` has block scope and `blue` has file scope:

```
int blue = 5;                /*  blue: file scope           */
main ()
{
    int x = 0 , y = 0;       /*  x and y: block scope       */
    int red = 10;            /*  red: block scope           */
    x = red + blue;
}
```

2.3.3 Function Scope

Only statement labels have *function scope* (see Chapter 7). An identifier with function scope is unique throughout the function in which it is declared. Labeled statements are used as targets for `goto` statements and are implicitly declared by their syntax, which is the label followed by a colon (:) and a statement. For example:

```
int func1(int x, int y, int z)
{
  label:  x += (y + z);   /*  label has function scope       */
  if (x > 1) goto label;
}
int func2(int a, int b, int c)
{
  if (a > 1) goto label; /*  illegal jump to undefined label */
}
```

See Section 7.1 for more information on statement labels.

2.3.4 Function Prototype Scope

An identifier that appears within a function prototype's list of parameter declarations has *function prototype scope*. The scope of such an identifier begins at the identifier's declaration and terminates at the end of the function prototype declaration list. For example:

```
int students ( int david, int susan, int mary, int john );
```

In this example, the identifiers (`david`, `susan`, `mary`, and `john`) have scope beginning at their declarations and ending at the closing parenthesis. The type of the function `students` is ''function returning `int` with four `int` parameters.'' In effect, these identifiers are merely placeholders for the actual parameter names to be used after the function is defined.

2.4 Visibility

An identifier is *visible* only within a certain region of the program. An identifier has *visibility* over its entire scope, unless a subsequent declaration of the same identifier in an enclosed block overrides, or *hides*, the previous declaration. Visibility affects the ability to access a data object or other identifier, because an identifier can be used only where it is visible.

Once an identifier is used for a specific purpose, it cannot be used for another purpose within the same scope, unless the second use of the identifier is in a different *name space*. Section 2.15 describes the name space restrictions. For example, declarations of two different data objects using the same name as an identifier is illegal within the same scope.

When the scope of one of two identical identifiers is contained within the other (nested), the identifier with inner scope remains visible, while the identifier with wider scope becomes hidden for the duration of the inner identifier's scope.

In the following example, the identifier `number` is used twice: once as an integer variable and once as a floating-point variable. For the duration of the function `main`, the integer `number` is hidden by the floating-point `number`.

```
#include <math.h>
int number;        /*  number is declared as an integer variable  */

main ()
{
 float x;
 float number;     /*  This declaration of number occurs in an inner
                       block, and "hides" the outer declaration.
                       The inner declaration creates a new object */
 x = sqrt (number);/*  x receives a floating-point value         */
}
```

2.5 Side Effects and Sequence Points

The actual order in which expressions are evaluated is not specified for most of the operators in C. Because this sequence of evaluation is determined within the compiler depending on context, some unexpected results may occur when using certain operators. These unexpected results are caused by *side effects*.

Any operation that affects an operand's storage has a side effect. Side effects can be deliberately induced by the programmer to produce a desired result; in fact, the assignment operator depends on the side effect of altered storage to do its job. C guarantees that all side effects of a given expression will be completed by the next sequence point in the program. *Sequence points* are

checkpoints in the program at which the compiler ensures that operations in an expression are concluded.

The most important sequence point is the semicolon marking the end of a statement. All expressions and their side effects are completely evaluated when the semicolon is reached. Other sequence points are as follows:

- *expr1*, *expr2* (the comma operator)
- *expr1* `&&` *expr2* (the logical AND operator)
- *expr1* `||` *expr2* (the logical OR operator)
- *expr1* ? *expr2* : *expr3* (the conditional operator)

These operations do guarantee the order, or sequence, of evaluation (*expr1*), *expr2*, and *expr3* are expressions). For each of these operators, the evaluation of expression *expr1* is guaranteed to occur before the evaluation of expression *expr2* (or *expr3*, in the case of the conditional expression).

Relying on the execution order of side effects, when none is guaranteed, is a risky practice because results are inconsistent and not portable. Undesirable side effects usually occur when the same data object is used in two or more places in the same expression, where at least one use produces a side effect. For example, the following code fragment produces inconsistent results because the order of evaluation of operands to the assignment operator is undefined.

```
int x[4] = { 0, 0, 0, 0 };
int i = 1;
x[i] = i++;
```

If the increment of `i` occurs before the subscript is evaluated, the value of `x[2]` is 1. If the subscript is evaluated first, the value of `x[1]` is 1.

A function call also has side effects. In the following example, the order in which `f1(y)` and `f2(z)` are called is undefined:

```
int y = 0;
int z = 0;
int x = 0;

int f1(int s)
   {
     printf ("Now in f1\n");
     y += 7;        /*  Storage of y affected   */
     return y;
    }
```

```
int f2(int t)
   {
     printf ("Now in f2\n");
     z += 3;        /*  Storage of z affected   */
     return z;
    }

main ()
{
x = f1(y) + f2(z);     /*  Undefined calling order   */
}
```

The `printf` functions can be executed in any order even though the value of `x` will always be 10.

2.6 Incomplete Type

An identifier can be initially declared as having an *incomplete type*. An incomplete type declaration describes the object, but lacks the information needed to determine the object's size. For example, a declaration of an array of unknown size is an incomplete type declaration:

```
extern int x[];
```

The incomplete type may be completed in a subsequent declaration. Incomplete types are most commonly used when forward referencing arrays, structures, and unions. (Section 2.12 discusses forward references.) An object of an aggregate type cannot contain a member of an incomplete type; therefore, an aggregate object (a structure or array member) cannot contain itself, because the aggregate type is not complete until the end of its declaration. The following example shows how an incomplete structure type is declared and later completed:

```
struct s
  { struct t *pt };  /* Incomplete structure declaration  */
.
.
.
struct t
   { int a;
     float *ps };    /*  Completion of structure t        */
```

The `void` type is a special case of an incomplete type. It is an incomplete type that cannot be completed, and is used to signify that a function returns no value. Section 3.5 has more information on the `void` type.

2.7 Compatible and Composite Types

Compatibility between types refers to the similarity of two types to each other. Type compatibility is important during type conversions and operations. All valid declarations in the same scope that refer to the same object or function must have compatible types. Two types are compatible if they fit any of the following categories:

- Two types are compatible if they are the same.
- Two qualified types (see Section 3.7) are compatible if they are identically qualified and the two types, unqualified, are compatible. The order of the qualifiers in the type declaration does not matter.
- The types `short`, `signed short`, `short int`, and `signed short int` are the same and are compatible.
- The types `unsigned short` and `unsigned short int` are the same and are compatible.
- The types `int`, `signed`, and `signed int` are the same and are compatible.
- The types `unsigned` and `unsigned int` are the same and are compatible.
- The types `long`, `signed long`, `long int`, `signed long int` are the same and are compatible.
- The types `unsigned long` and `unsigned long int` are the same and are compatible.
- Two array types are compatible if they are of the same size and contain elements of compatible types. If one array has an unknown size, it is compatible with all other array types having compatible element types.
- Two unions or structures are compatible if they are declared in different compilation units, share the same members in the same order, and whose members have the same widths (including bit fields).
- Two enumerations are compatible if all members have the same values. All enumerated types are compatible with other enumerated types. An enumerated type is also compatible with the `signed int` type.
- Two pointer types are compatible if they are identically qualified and point to objects of compatible types.
- A function type declared using the old-style declaration (such as `int tree()`) is compatible with another function type if the return types are compatible.

- A function type declared using the new prototype-style declaration (such as `int tree (int x)`) is compatible with another function type declared with a function prototype if:
  - The return types are compatible.
  - The parameters agree in number (including an ellipsis if one is used).
  - The parameter types are compatible. For each parameter declared with a qualified type, its type for compatibility comparison is the unqualified version of the declared type.
- The function type of a prototype-style function declaration is compatible with the function type of an old-style function declaration if the return types are compatible, and if the old-style declaration is not a definition. (Different styles of function declarations are discussed in Chapter 5.) Otherwise, the function type of a prototype-style function declaration is compatible with the function type of an old-style function definition if all of the following conditions are met:
  - The return types of the two functions are compatible.
  - The number of parameters agree.
  - The prototype-style function declaration does not contain an ellipsis as a parameter.
  - The promoted types of the old-style parameters are compatible with the prototype-style parameter types. In the following example, the functions `tree` and `tree2` are compatible. `tree` and `tree1` are not compatible, and `tree1` and `tree2` are not compatible.

```
int tree (int);
int tree1 (char);
int tree2 (x)
   char x;   /* char promotes to int in old-style
                function parameters, and so is
                compatible with tree             */
   {
   ...
   };
```

The following types, which may appear to be compatible, are not:

- `unsigned int` and `int` types are not compatible.
- `char`, `signed char`, and `unsigned char` types are not compatible.

Composite Type

A *composite type* is constructed from two compatible types and is compatible with both of the two types. Composite types satisfy the following conditions:

- If one type is an array of known size, the composite type is an array of that size. Otherwise, if one type is a variable-length array, the composite type is that type.
- If only one type is a function type with a prototype, the composite type is a function type with the parameter type list.
- If both types are functions types with prototypes, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

Consider the following file-scope declarations:

```
int f(int (*) (), double (*) [3]);
int f(int (*) (char *), double (*)[]);
```

They result in the following composite type for the function:

```
int f(int (*) (char *), double (*)[3]);
```

The previous composite type rules apply recursively to types derived from composite types.

2.8 Linkage

Data objects and functions can be implicitly or explicitly assigned *linkage*. There are three kinds of linkage:

- Internal linkage—a declaration referring to a data object or function declared in the same compilation unit, and not known outside the compilation unit.
- External linkage—a declaration referring to a definition of a data object or function known outside the compilation unit. The definition of the object also has external linkage.
- No linkage—a declaration declaring a unique data object.

When more than one declaration of the same object or function is made, linkage is made. The linked declarations can be in the same scope or in different scopes. Externally linked objects are available to any function in any compilation unit used to create the executable file. Internally linked objects are available only to the compilation unit in which the declarations appear.

The concept of linkage and the `static` and `extern` keywords are related, but not directly. Using the `extern` keyword in an object's declaration does not guarantee external linkage. The following rules determine the actual linkage of an object or function:

- An identifier explicitly specified with the `auto` or `register` storage class has no linkage.
- An identifier with block scope and the `extern` storage-class specification has linkage the same as any visible declaration of the same identifier with file scope. If no such declaration of the object or function is visible, then the object or function has external linkage.
- The declaration of functions defaults to external linkage. The only other storage class possible for a function is `static`, which must be specified explicitly, cannot be applied to a block scope function declaration, and results in internal linkage.
- The file scope declaration of a data object without an explicit storage class specification, or with the `extern` storage class specified, has external linkage.
- An identifier with file scope and the `static` storage class has internal linkage.
- An identifier with block scope and without the `extern` storage-class specification has no linkage.

Identifiers other than data objects and functions have no linkage. An identifier declared as a function parameter also has no linkage.

The following examples show declarations with different linkages:

```
extern int x;          /*  External linkage                      */
static int y;          /*  Internal linkage                      */
register int z;        /*  Illegal storage-class declaration     */

main ()                /*  Functions default to external linkage */
{
    int w;             /*  No linkage                            */
    extern int x;      /*  External linkage                      */
    extern int y;      /*  Internal linkage                      */
    static int a;      /*  No linkage                            */
}

void func1 (int arg1)  /*  arg1 has no linkage                   */
{ }
```

In HP C, a message is issued if the same object is declared with both internal and external linkage.

2.9 Tentative Definitions

A declaration of an identifier with file scope, no initializer, and either no storage-class specifier or the `static` storage-class specifier is a *tentative definition*. The tentative definition only applies if no other definition of the object appears in the compilation unit, in which case all tentative definitions for an object are treated as if there were only one file scope definition of the object, with an initializer of zero.

If a definition for a tentatively defined object is used later in the compilation unit, the tentative definition is treated as a redundant declaration of the object. If the declaration of an identifier for an object is a tentative definition and has internal linkage, the declared type cannot be an incomplete type. Section 2.8 discusses linkage.

The following are examples of tentative definitions:

```
int i1 = 1;    /* Standard definition with external linkage      */
int i4;        /* Tentative definition with external linkage     */
static int i5; /* Tentative definition with internal linkage     */
int i1;        /* Valid tentative definition, refers to previous */
               /* i1 declaration                                 */
```

2.10 Storage Classes

Storage classes apply only to data objects and function parameters. However, storage class keywords in C are also used to affect the visibility of functions. Every data object and parameter used in a program has exactly one storage class, either assigned explicitly or by default. There are four storage classes:

- `auto`
- `register`
- `static`
- `extern`

An object's storage class determines its availability to the linker and its *storage duration*. An object with external or internal linkage, or with the storage-class specifier `static`, has static storage duration, which means that storage for the object is reserved and initialized to 0 only once, before `main` begins execution. An object with no linkage and without the storage-class specifier `static` has automatic storage duration; for such an object, storage is automatically allocated on entry to the block in which it is declared, and automatically deallocated on exiting from the block. An automatic object is not initialized.

When applied to functions, the storage-class specifier `extern` makes the function visible from other compilation units, and the storage-class specifier `static` makes the function visible only to other functions in the same compilation unit. For example:

```
static int tree(void);
```

The following sections describe these storage classes.

2.10.1 The auto Class

The `auto` class specifies that storage for an object is created upon entry to the block defining the object, and destroyed when the block is exited. This class can be declared only at the beginning of a block, such as at the beginning of a function's body. For example:

```
auto int a;        /*  Illegal -- auto must be within a block  */

main ()
{
    auto int b;               /*  Valid auto declaration   */
    for (b = 0; b < 10; b++)
      {
        auto int a = b + a;   /*   Valid inner block declaration    */
      }
}
```

When you use an initializer with an `auto` object (see Section 4.2), the object is initialized each time it is created. Storage is reserved for the object whether the block containing the object is entered through normal processing of the block or through a jump statement into the block. However, if the block is entered through a jump statement, initialization of the object is not guaranteed, and if the object is a variable-length array, storage is not reserved.

The `auto` class is the default for objects with block scope. Objects with the `auto` class are not available to the linker.

Note

Entering an enclosed block suspends, but does not end, execution of the enclosing block. Calling a function from within a block suspends, but does not end, execution of the block containing the call. Automatic objects with reserved storage maintain their storage in these cases.

2.10.2 The register Class

The `register` class identifies the assigned object as frequently used, suggesting to the compiler that the object should be assigned a register to minimize access time. `register` is never the default class; it must be explicitly specified.

The `register` class has the same storage duration as the `auto` class; that is, storage is created for a `register` object upon entry to the block defining the object, and destroyed when the block is exited.

The `register` class is the only storage class that can be explicitly specified for function parameters.

The HP C compiler uses sophisticated register allocation techniques that make the use of the `register` keyword unnecessary.

2.10.3 The static Class

The `static` class specifies that space for the identifier is maintained for the duration of the program. Static objects are not available to the linker. Therefore, another compilation unit can contain an identical declaration that refers to a different object.

A `static` object can be declared anywhere a declaration may appear in the program; it does not have to be at the beginning of a block, as with the `auto` class. If a data object is declared outside a function, it has static duration by default—it is initialized only once at the beginning of the program.

Expressions used to initialize `static` objects must be constant expressions. If the object with `static` storage duration is not explicitly initialized, every arithmetic member of that object is initialized to 0, and every pointer member is initialized as a null pointer constant. See Section 4.2 for more information on initializing objects of various data types.

2.10.4 The extern Class

The `extern` class is the default class for objects with file scope. Objects outside of any function (an external definition) receive the `extern` class storage unless explicitly assigned the `static` keyword in the declaration. The `extern` class specifies the same storage duration as `static` objects, but the object or function name is *not* hidden from the linker. Using the `extern` keyword in a declaration results in external linkage in most cases (see Section 2.8), and results in static duration of the object.

2.11 Storage-Class Modifiers

HP C provides the following storage-class modifiers:

```
__inline
__forceinline
__align
inline
```

The first three modifiers listed are recognized as valid keywords in all compiler modes on all platforms. They are in the namespace reserved to the C implementation, so it is not necessary to allow them to be treated as user-declared identifiers. They have the same effects on all platforms, except that on OpenVMS VAX systems, the `__forceinline` modifier does not cause any more inlining than the `__inline` modifier does.

The `inline` storage-class modifier is supported in relaxed ANSI C mode or if the /ACCEPT=C99_KEYWORDS *(OpenVMS)* or /ACCEPT=GCCINLINE *(OpenVMS)* qualifier is specified.

Note

HP C for *OpenVMS Systems* also provides support for the storage-class modifiers `noshare`, `readonly`, and `_align` as VAX C keywords. For more information about these storage-class modifiers, see the *HP C User's Guide for OpenVMS Systems* *(OpenVMS)*.

You can use a storage-class specifier and a storage-class modifier in any order. Usually, the modifier is placed after the specifier in the source code. For example:

```
extern  noshare  int  x;
   /*  Or, equivalently . . .    */
int  noshare  extern  x;
```

However, placing the storage-class specifier anywhere other than first is obsolescent.

The following sections describe each of the HP C storage-class modifiers.

2.11.1 The __inline Modifier

The `__inline` storage-class modifier marks a function for inline expansion. Using `__inline` on a function definition and prototype tells the compiler that it can substitute the code within the function definition for every call to that function. Substitution occurs at the discretion of the compiler. The `__inline` storage-class modifier has the same effect as the `#pragma inline` preprocessor directive, except that `#pragma inline` attempts to provide inline expansion for all functions in a translation unit, rather than for selected functions (See your platform-specific HP C documentation for more information on `#pragma inline`).

Use the following form to designate a function for inline expansion:

__inline [*type*] *function_definition*

The compiler issues a warning if `__inline` is used in /STANDARD=PORTABLE mode, because this is an implementation-specific extension.

Here is an example of using `__inline`:

```
/* prototype */

__inline int x (float y);

/* definition */

__inline int x (float y)

{
   return (1.0);
}
```

2.11.2 The inline Modifier

Similar to the `__inline` storage-class modifier, the `inline` storage-class modifier can be used as a declaration specifier in the declaration of a function.

The `inline` storage-class modifier is supported in relaxed ANSI C mode or if the /ACCEPT=C99_KEYWORDS *(OpenVMS)* or /ACCEPT=GCCINLINE *(OpenVMS)* qualifier is specified.

With static functions, `inline` has the same effect as applying `__inline` or `#pragma inline` to the function.

However, when `inline` is applied to a function with external linkage, besides allowing calls within that translation unit to be inlined, the `inline` semantics provide additional rules that also allow calls to the function to be inlined in other translation units or for the function to be called as an external function, at the compiler's discretion:

- If the `inline` keyword is used on a function declaration with external linkage, then the function must also be defined in the same translation unit.
- If all of the file-scope declarations of the function use the `inline` keyword but do not use the `extern` keyword, then the definition in that translation unit is called an inline auxiliary definition, and no externally-callable (global) definition is produced by that compilation unit.

  Otherwise, the compilation unit does produce an externally-callable definition.
- An `inline` auxiliary definition must not contain a definition of a modifiable object with static storage duration, and it must not refer to an identifier with internal linkage. These restrictions do not apply to the externally-callable definition.
- As usual, at most one compilation unit in an entire program can supply an externally-callable definition of a given function.
- Any call to a function with external linkage might be translated as a call to an external function, regardless of the presence of the `inline` qualifier. It follows from this and the previous point that any function with external linkage that is called must have exactly one externally-callable definition among all the compilation units of an entire program.
- The address of an `inline` function with external linkage is always computed as the address of the unique externally-callable definition, never the address of an inline definition.
- A call to an `inline` function made through a pointer to the externally-callable definition may still be inlined or translated as a call to an inline definition, if the compiler can determine the name of the function whose address was stored in the pointer.
- Without the `inline` keyword, a function definition in a header file produces MULDEF errors at link time, if the header file is included by more than one translation unit. Specifying `inline` on such a function definition is one way to eliminate these MULDEF errors. See the example (Section 2.11.2.1).

Note

This section describes the semantics of the C99 Standard `inline` keyword.

The gcc compiler implements an inline function declaration specifier for functions with external linkage that gives similar capabilites to this

C99 inline feature, but the details of usage are somewhat different: essentially, the combination of `extern` and `inline` keywords makes an inline definition, instead of the exclusive use of the `inline` keyword without the `extern` keyword.

The /ACCEPT=[NO]GCCINLINE qualifier controls which variation of the feature is implemented.

2.11.2.1 Example—Using the inline Function Specifier

Consider the following C code, which results in a multiply defined function identifier (`my_max`):

```
$ type t.h
int my_max (int x, int y)
{
    if (x >= y)
        return (x);
    else
        return (y);
}
$
$ type a.c
#include "t.h"

main()
{
    int a =1;
    int b=2;

    func1();
    my_max(func1(a,b),20);
}
$
$ type b.c
#include "t.h"

void func1(int p1, int p2)
{
    my_max(p1,p2);
}
$
$ link a,b
%LINK-W-MULDEF, symbol MY_MAX multiply defined
        in module B file DISK$:[TEST.TMP]B.OBJ;4
```

One way around this problem is to define the function my_max with the keyword static:

```
static int my_max (int x, int y)
{
    if (x >= y)
        return (x);
    else
        return (y);
}
```

However, this means there is no globally visible my_max function but, rather, a copy of my_max for each module, each copy with a different address. Therefore, any function pointer comparisons would break.

The ISO C99 solution to this problem is the inline keyword. Adding inline to the header file t.h eliminates the MULDEF errors:

```
inline int my_max (int x, int y)
{
    if (x >= y)
        return (x);
    else
        return (y);
}
```

This type of function definition, like one specified with the __inline keyword, marks the function for potential inlining by the compiler. One difference, however, is that for an inline function, the compiler creates an inline auxiliary definition of the function, which is associated with the function being declared (my_max in this example). The compiler is then free to do one of the following:

1. Call the auxiliary function.
2. Call the global function (my_max). This implies that there *must* be a global definition of any non-static inline function in one of the modules of the application.
3. Generate inlined code for the call to my_max.

There can be one and only one global definition for the inline function within an application. There can be one inline auxiliary definition per module, or many prototype declarations of the auxiliary function per module.

You can create a global inline definition by including in one of your modules (such as a.c in our example) a file-scope function declaration that:

1. Omits the inline keyword:

```
#include "t.h"
int my_max (int x, int y);
```

OR

```
#include "t.h"
extern int my_max (int x, int y);
```

2. Or that specifies the `extern` storage class with the `inline` keyword:

```
#include "t.h"
extern inline int my_max (int x, int y);
```

Note

Taking the address of an `inline` function always resolves to the global function, never the auxiliary function.

2.11.3 The __forceinline Modifier

Similar to the `__inline` storage-class modifier, the `__forceinline` storage-class modifier marks a function for inline expansion. However, using `__forceinline` on a function definition and prototype tells the compiler that it *must* substitute the code within the function definition for every call to that function. (With `__inline`, such substitution occurs at the discretion of the compiler.)

On OpenVMS VAX systems, the `__forceinline` storage-class modifier does not cause any more inlining to occur than the `__inline` modifier does.

Use the following form to designate a function for forced inline expansion:

__forceinline [*type*] *function_definition*

The compiler issues a warning if `__forceinline` is used in /STANDARD=PORTABLE mode, because this is an implementation-specific extension.

2.11.4 The __align Modifier

The `__align` and `_align` storage-class modifiers have the same semantic meaning. The difference is that `__align` is a keyword in all compiler modes while `_align` is a keyword only in modes that recognize VAX C keywords. For new programs, using `__align` is recommended.

The `__align` storage-class modifier aligns objects of any of the HP C data types on a specified storage boundary. Use the `__align` modifier in a data declaration or definition.

For example, to align an integer on the next quadword boundary, you can use any of the following declarations:

```
int  __align( QUADWORD )  data;
int  __align( quadword )  data;
int  __align( 3 )  data;
```

When specifying the boundary of the data alignment, you can either use a predefined constant or specify an integer value that is a power of 2. These constants, or explicit powers of 2, tell HP C the number of bytes to pad in order to align the data. In the previous example, `int __align ( 3 )` specifies an alignment of 2^3 bytes, which is 8 bytes—a quadword of memory.

Table 2–1 presents all the predefined alignment constants, their equivalent power of 2, and their equivalent number of bytes. Note that for OpenVMS VAX systems, you can specify the constants 0, 1, 2, 3, 4, or 9. For OpenVMS Alpha systems, you can specify any constant from 0 to 16.

Table 2–1 Predefined Alignment Constants

Constant	Power of 2	Number of Bytes
BYTE or byte	0	1
WORD or word	1	2
LONGWORD or longword	2	4
QUADWORD or quadword	3	8
OCTAWORD or octaword	4	16
	5 *(Alpha, I64)*	32
	6 *(Alpha, I64)*	64
	7 *(Alpha, I64)*	128
	8 *(Alpha, I64)*	256
	9	512
	10 *(Alpha, I64)*	1024
	11 *(Alpha, I64)*	2048

(continued on next page)

Table 2–1 (Cont.) Predefined Alignment Constants

Constant	Power of 2	Number of Bytes
	12 *(Alpha, I64)*	4096
	13 *(Alpha, I64)*	8192
	14 *(Alpha, I64)*	16384
	15 *(Alpha, I64)*	32768
PAGE or page	16 *(Alpha, I64)* 9 *(VAX only)*	65,536 *(Alpha, I64)* 512 *(VAX only)*

2.12 Forward References

Once declared, identifiers can be used freely. Using an identifier *before* its declaration is called a *forward reference*, and results in an error, except in the following cases:

- When a `goto` statement refers to a statement label before the label's declaration
- When a structure, union, or enumeration tag is used before it is declared

Here are some examples of valid and invalid forward references:

```
int a;
main ()
{
 int b = c;           /*  Forward reference to c -- illegal        */
 int c = 10;
 glop x = 1;          /*  Forward reference to glop type -- illegal */
 typedef int glop;
 goto test;           /*  Forward reference to statement label --
                          legal                                    */
test:
 if (a > 0 ) b = TRUE;
}
```

The following example shows the use of a structure tag in a forward reference:

```
struct s
  { struct t *pt };    /*  Forward reference to structure t         */
.                      /* (Note that the reference is preceded      */
.                      /* by the  struct keyword to resolve         */
.                      /* potential ambiguity)                      */
struct t
   { struct s *ps };
```

2.13 Tags

Tags can be used with structures, unions, or enumerated types as a means of referring to the structure, union, or enumerated type elsewhere in the program. Once a tag is included in the declaration of a structure, union, or enumerated type, it can specify the declared structure, union, or enumerated type anywhere the declaration is visible.

The following code fragment shows the use of a structure tag, a union tag, and an enumerated type tag:

```
struct tnode {                  /*  Initial declaration --             */
                                /*  tnode is the structure tag         */
 int count;
 struct tnode *left, *right;    /*  tnode's members referring to tnode */
 union datanode *p;             /*  forward reference to union type is
                                    declared below                     */
};

union datanode {                /*  Initial declaration --             */
                                /*  datanode is the union tag          */
 int ival;
 float fval;
 char *cval;
} q = {5};

enum color { red, blue, green };/*  Initial declaration --             */
.                                /*  color is the enumeration tag      */
.
.
struct tnode x;                 /*   tnode tag is used to declare x    */
enum color z = blue;            /*   color tag declares z to be of
                                     type color;  z is also
                                     initialized to blue               */
```

As shown in the previous example, once a tag is declared it can be used to reference other structure, union, or enumerated type declarations in the same scope without fully redefining the object.

Tags can be used to form an incomplete type if they occur before the complete declaration of a structure or union. Incomplete types do not specify the size of the object; therefore, a tag introducing an incomplete type can only be used when the size of the object is not needed. To complete the type, another declaration of the tag in the same scope must define the object completely. The following example shows how a subsequent definition completes the incomplete declaration of the structure type `s`:

```
struct s;            /*  Tag s used in incomplete type declaration */
struct t {
  struct s *p;
};
struct s { int i; };/*  struct s definition completed              */
```

Section 2.6 describes the concept of an incomplete type.

Consider the following declarations:

```
struct tag;
```

```
union tag;
```

These declarations specify a structure or union type and declare a tag visible only within the scope of the declaration. The declaration specifies a new type distinct from any other type with the same tag in an enclosing scope (if any).

The following example shows the use of prior tag declarations to specify a pair of mutually-referential structures:

```
struct s1 { struct s2 *s2p; /*...*/ };  /* D1  */
struct s2 { struct s1 *s1p; /*...*/ };  /* D2  */
```

If `s2` was declared as a tag in an enclosing scope, the declaration `D1` would refer to `s2`, not to the tag `s2` declared in `D2`. To eliminate this context sensitivity, the following declaration can be inserted ahead of `D1`:

```
struct s2;
```

This declares a new tag `s2` in the inner scope; the declaration `D2` then completes the specification of the type.

2.14 lvalues and rvalues

An *rvalue* is the value of an expression, such as 2, or $x + 3$, or $(x + y) * (a - b)$. rvalues are not allocated storage space. Examples of rvalues are the numbers 0 and 1 in the following code fragment:

```
if (x > 0)
   {
     y += 1;
   }
x = *y;        /*  The value pointed to by y is assigned to x   */
```

The identifiers `x` and `y` are objects with allocated storage. The pointer to `y` holds an lvalue.

An *lvalue* is an expression that describes the location of an object used in the program. The location of the object is the object's lvalue, and the object's rvalue is the value stored at the location described by the lvalue. The following operators always produce lvalues:

```
[]
*
->
```

The dot operator (.) can, and usually does, produce an lvalue but it does not have to do so. For example, `f().m` is not an lvalue.

A *modifiable lvalue* is an lvalue that does not have array type, an incomplete type, a `const`-qualified type, or, if it is a structure or union, has no member with `const`-qualified type.

2.15 Name Spaces

Name spaces are identifier classifications based on the context of the identifier's use in the program. Name spaces allow the same identifier to simultaneously stand for an object, statement label, structure tag, union member, and enumeration constant. Simultaneous use of an identifier in the same scope for two different entities without ambiguity is possible only if the identifiers are in different name spaces. The context of the identifier's use resolves the ambiguity over which of the identically named entities is desired.

There are four different name spaces:

- Statement labels
- Structure, union, and enumeration tags
- Each structure and union member set
- Other identifiers (variables, functions, type definitions, and enumeration constants)

For example, the identifier `flower` can be used in one block to stand for both a variable and an enumeration tag, because variables and tags are in different name spaces. Subsequently, an inner block can redefine the variable `flower` without disturbing the enumeration tag `flower`. Therefore, when using the same identifier for various purposes, analyze the name space and scope rules governing the identifier. Section 2.3 presents the scope rules.

A structure, union, and enumeration member name can be common to each of these objects at the same time. The use of the structure, union, or enumeration name in the reference to the member resolves any ambiguity about which identifier is meant. However, the structure, union, or enumeration

tag must be unique, since the tags of these three object types share the same name space.

2.16 Preprocessing

The translation of a C program occurs in several phases. Normally, when the compiler is started, several events occur before the actual compiler starts:

1. Trigraph sequences (if any) are replaced by single-character internal representations.
2. Each occurrence of a new-line character immediately preceded by a backslash character is deleted and the following line is spliced to form one logical line.
3. The source file is decomposed into preprocessing tokens and sequences of white-space characters. Each comment is replaced by one space character.
4. Preprocessing directives are executed and preprocessor macros are expanded. Files named in `#include` preprocessing directives are processed through these four steps recursively.
5. Each source character set member, and each escape sequence in character constants and string literals is converted to a member of the execution character set.
6. Adjacent character string literal tokens are concatenated and adjacent wide string literal tokens are concatenated.
7. The resulting stream of tokens is analyzed and translated.
8. The linking phase. All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current compilation unit. All such linker output is collected into a program image.

The fourth step is called *preprocessing*, and is handled by a separate unit of the compiler. Each preprocessor directive appears on a line beginning with a pound sign (#); white space may precede the pound sign. These lines are syntactically independent from the rest of the C source file, and can appear anywhere in the source file. Preprocessor directive lines terminate at the end of the logical line.

It is possible to preprocess a source file without actually compiling the program (see your platform-specific HP C documentation for the available compiler options.) Chapter 8 discusses the preprocessing directives.

2.17 Type Names

In several contexts a type name can or must be specified without an identifier. For example, in a function prototype declaration, the parameters of the function can be declared only with a type name. Also, when casting an object from one type to another, a type name is required without an associated identifier. (Section 6.4.6 has information on casting, and Section 5.5 has information on function prototypes.) This is accomplished using a *type name*, which is a declaration for a function or object which omits the identifier.

Table 2–2 shows examples of type names with the associated types they refer to.

Table 2–2 Type Name Examples

Construction	Type Name
`int`	`int`
`int *`	Pointer to `int`
`int *[3]`	Array of three pointers to `int`
`int (*)[3]`	Pointer to an array of three `int`s
`int *()`	Function with no parameter specification returning a pointer to `int`
`int (*) (void)`	Pointer to function with no parameters returning an `int`
`int (*const []) (unsigned int, ...)`	Array of an unspecified number of `const` pointers to functions, each with one parameter that has type `unsigned int` and an unspecified number of other parameters, returning an `int`

Table 2–2 also provides good examples of *abstract declarators*. An abstract declarator is a declarator without an identifier. The characters following the `int` type name form an abstract declarator in each case. The `*`, `[ ]`, and `( )` characters all indicate a declarator without naming a specific identifier.

3
Data Types

The *type* of a data object in C determines the range and kind of values an object can represent, the size of machine storage reserved for an object, and the operations allowed on an object. Functions also have types, and the function's return type and parameter types can be specified in the function's declaration.

The following sections discuss these topics:

- Data sizes (Section 3.1)
- Integral types (Section 3.2)
- Floating-point types (Section 3.3)
- Derived types (Section 3.4), including:
  - Function type (Section 3.4.1)
  - Pointer type (Section 3.4.2)
  - Array type (Section 3.4.3)
  - Structure type (Section 3.4.4)
  - Union type (Section 3.4.5)
- `void` type (Section 3.5)
- Enumerated types (Section 3.6)
- Type qualifiers (Section 3.7)
- Type definition (Section 3.8)

The selection of a data type for a given object or function is one of the fundamental programming steps in any language. Each data object or function in the program must have a data type, assigned either explicitly or by default. (Chapter 4 discusses the assignment of a data type to an object.) C offers a wide variety of types. This diversity is a strong feature of C, but can be initially confusing.

To help avoid this confusion, remember that C has only a few basic types. All other types are derived combinations of these basic types. Some types can be specified in more than one way; for example, `short` and `short int` are the same type. (In this manual, the longest, most specific name is always used.) Type is assigned to each object or function as part of the *declaration*. Chapter 4 describes declarations in more detail.

Table 3–1 lists the *basic data types*: *integral types* (objects representing integers within a specific range), *floating-point types* (objects representing numbers with a significand part—a whole number plus a fractional number—and an optional exponential part), and *character types* (objects representing a printable character). Character types are stored as integers.

Note

Enumerated types are also normally classified as integral types, but for the purposes of clarity they are not listed here. See Section 3.6 for more information.

Table 3–1 Basic Data Types

Integral Types	Floating Point Types
`short int`	`float`
`signed short int`	`double`
`unsigned short int`	`long double`
`int`	`float _Complex` *(Alpha, I64)*
`signed int`	`double _Complex` *(Alpha, I64)*
`unsigned int`	`long double _Complex` *(Alpha, I64)*
`_Imaginary` In HP C, use of the `_Imaginary` keyword produces a warning, which is resolved by treating it as an ordinary identifier.	
`long int`	
`signed long int`	
`unsigned long int`	
`long long int` *(Alpha, I64)*	

(continued on next page)

Table 3–1 (Cont.) Basic Data Types

Integral Types	Floating Point Types
`signed long long int` *(Alpha, I64)*	
`unsigned long long int` *(Alpha, I64)*	
`_Bool`	

Integral Character Types
`char`
`signed char`
`unsigned char`

The integral and floating-point types combined are called the *arithmetic types*. See Section 3.1 for information about the size and range of integral and floating-point values.

A large variety of *derived* types can be created from the basic types. Section 3.4 discusses the derived types.

Besides the basic and derived types, there are three keywords that specify unique types: `void`, `enum`, and `typedef`:

- The `void` keyword specifies a special type indicating no value, or it can be used with the pointer operator (*) to indicate a generic pointer type. See Section 3.5 for more information on the `void` type.
- The `enum` keyword specifies an integer type of your own design, specifying the acceptable values of the type to a predefined set of named integer constant values. Enumerated types are stored as integers, except in the compiler's RELAXED mode, in which other types are allowed. See Section 3.6 for a detailed description of enumerated types.
- The `typedef` keyword specifies a synonym for a type made from one or more basic or derived types. See Section 3.8 for more information on creating type definitions.

There are also the *type-qualifier* keywords:

- `const`, used to prevent write access to an object (see Section 3.7.1)
- `volatile`, used to restrict the optimizations that might otherwise be performed on references to an object (see Section 3.7.2)

- `__unaligned` *(Alpha, I64)*, used in pointer definitions, to indicate to the compiler that the data pointed to is not properly aligned on a correct address
- `__restrict` (for pointer type only), used to designate a pointer as pointing to a distinct object, thus allowing compiler optimizations to be made (see Section 3.7.4)

Using a qualifying keyword in the type declaration of an object results in a *qualified type*. See Section 3.7 for general information on type qualifiers.

With such a wide variety of types, operations in a program often need to be performed on objects of different types, and parameters of one type often need to be passed to functions expecting different parameter types. Because C stores different kinds of values in different ways, a *conversion* must be performed on at least one of the operands or arguments to convert the type of one operand or argument to match that of the other. You can perform conversions explicitly through *casting*, or implicitly through the compiler. See Section 6.11 for more information on data-type conversions. See Section 2.7 for a description of type compatibility.

See your platform-specific HP C documentation for a description of any implementation-defined data types.

3.1 Data Sizes

An object of a given data type is stored in a section of memory having a discreet size. Objects of different data types require different amounts of memory. Table 3–2 shows the size and range of the basic data types.

Table 3–2 Sizes and Ranges of Data Types

Type	Size	Range
Integral Types		
`short int`, or `signed short int`	16 bits	−32768 to 32767
`unsigned short int`	16 bits	0 to 65535
`int` or `signed int`	32 bits	−2147483648 to 2147483647
`unsigned int`	32 bits	0 to 4294967295

(continued on next page)

Table 3–2 (Cont.) Sizes and Ranges of Data Types

Type	Size	Range
Integral Types		
`long int`, or `signed long int` *(OpenVMS)*	32 bits	−2147483648 to 2147483647
`long int`, or `signed long int` *(Tru64 UNIX)*	64 bits	−9223372036854775808 to 9223372036854775807
`unsigned long int` *(OpenVMS)*	32 bits	0 to 4294967295
`unsigned long int` *(Tru64 UNIX)*	64 bits	0 to 18446744073709551615
`signed long long int` *(Alpha, I64)*, `signed __int64` *(Alpha, I64)*	64 bits	−9223372036854775808 to 9223372036854775807
`unsigned long long int` *(Alpha, I64)*, `unsigned __int64` *(Alpha, I64)*	64 bits	0 to 18446744073709551615
Integral Character Types		
`char` and `signed char`	8 bits	−128 to 127
`unsigned char`	8 bits	0 to 255
`wchar_t`	32 bits	0 to 4294967295
Floating-Point Types (range is for absolute value)		
`float`	32 bits	1.1×10^{-38} to 3.4×10^{38}
`double`	64 bits	2.2×10^{-308} to 1.7×10^{308}
`long double` *(OpenVMS Alpha, I64)*	128 bits	3.4×10^{-49321} to 1.2×10^{1049321}
`long double` *(OpenVMS VAX, Tru64 UNIX)*	Same as `double`	Same as `double`

Derived types can require more memory space.

See your platform-specific HP C documentation for the sizes of implementation-defined data types.

3.2 Integral Types

In C, an integral type can declare:

- Integer values, signed or unsigned
- Boolean values, where 0 is equivalent to false and any nonzero number is equivalent to true
- Characters, which are automatically converted to an integer value by the compiler
- Members of an enumerated type, which are interpreted as an integer by the compiler
- Bit fields

The integral types are:

- `char`, `signed char`, `unsigned char`—8 bits
- `short int`, `signed short int`, and `unsigned short int`—16 bits
- `_Bool`—1 byte
- `int`, `signed int`, `unsigned int`—32 bits
- `long int`, `signed long int`, and `unsigned long int`—32 bits *(OpenVMS)*
- `long int`, `signed long int`, and `unsigned long int`—64 bits *(Tru64 UNIX)*
- `signed long long int` *(Alpha, I64)* and `unsigned long long int` *(Alpha, I64)*—64 bits
- `signed __int64` *(Alpha, I64)* and `unsigned __int64` *(Alpha, I64)*—64 bits
- `enum`—32 bits

3.2.1 Non-Character Types

For HP C on OpenVMS systems, storage for `int` and `long` is identical. Similarly, storage of `signed int` and `signed long` is identical, and storage for `unsigned int` and `unsigned long` is identical.

For HP C on Tru64 UNIX systems, storage for the `int` data types is 32 bits, while storage for the `long int` data types is 64 bits.

The 64-bit integral types `signed long long int` and `unsigned long long int`, and their equivalents `signed __int64` and `unsigned __int64` are provided on Alpha and Itanium processors only. Note: the `__int64` and `long long int` data types (both signed and unsigned) can be used interchangeably, except for use with pointer operations, in which case the pointer types must be identical:

```
__int64 *p1;
__int64 *p2;
long long int *p3;
   .
   .
   .
p1 = p2;        // valid
p1 = p3;        // invalid
```

For each of the signed integral types, there is a corresponding *unsigned integral* type that uses the same amount of storage. The `unsigned` keyword with the integral type modifies the way the integer value is interpreted, which allows the storage of a larger range of positive values. When using the `unsigned` keyword, the bits are interpreted differently to allow for the increased positive range with the unsigned type (at the expense of the negative range of values). For example:

```
signed short int x = 45000;  /*  ERROR -- value too large for short int  */
unsigned short int y = 45000;/*  This value is OK                        */
```

The range of values for the `signed short int` type is −32,768 to 32,767. The range of values for the `unsigned short int` type is 0 to 65,535.

A computation involving unsigned operands can never overflow, because any result outside the range of the `unsigned` type is reduced to fit the type by the rules of modulus arithmetic. If the result cannot be represented by the resulting integer type, the result is reduced modulo the number that is one greater than the largest value that *can* be represented by the resulting unsigned integer type. This means that the low-order bits are kept, and the high-order bits of the mathematical result that do not fit in the type of the result are discarded. For example:

```
unsigned short int z = (99 * 99999); /*  Value of y after evaluation is 3965  */
```

HP C treats the plain `char` type as `signed` by default for compatibility with VAX C and many other C compilers. However, a command-line option can control this, and a predefined macro can be tested to determine the setting of the option in a given compilation. On Alpha systems, `unsigned char` might offer some performance advantage for character-intensive processing.

An unsigned integer of n bits is always interpreted in straight unsigned binary notation, with possible values ranging from 0 to $2^n - 1$.

Note

The interpretation of signed integers depends on the size of machine representation and the encoding technique used on the machine. With

two's-complement representation, signed integers of *n* bits have a range of -2^{n-1} to $2^{n-1} - 1$.

The C99-specified _Bool data type is available in all modes of the compiler except VAX C, common, and strict ANSI89 modes. A _Bool object occupies a single byte of storage and is treated as an unsigned integer, but its value can be only 0 or 1.

Notes

- A bit field can be declared to be type _Bool.
- A pointer can be converted to a _Bool type.
- When any scalar value is converted to _Bool, the result is 0 if the value compares equal to 0 (for example, if the pointer is NULL). Otherwise, the result is 1. This is one way the _Bool type is different than the other integer types. In the following example, the value of b is zero, but the value of c is 1:

```
double a = .01;
int b = a;
_Bool c = a;
```

- The _Bool type is intended to be used in conjuction with a new standard header, <stdbool.h>, but that is not required. The content of the new header is:

```
#define bool _Bool
#define true 1
#define false 0
#define __bool_true_false_are_defined 1
```

Also see Section 9.11.

3.2.2 Character Types

Character types are declared with the keyword char and are integral types. Using char objects for nonintegral operations is not recommended, as the results are likely to be nonportable. An object declared as a char type can always store the largest member of the source character set.

Valid character types are:

- char

- signed char
- unsigned char
- wchar_t

The *wide character* type wchar_t is provided to represent characters not included in the ASCII character set. The wchar_t type is defined using the typedef keyword in the <stddef.h> header file. Wide characters used in constants or strings must be preceded with an L. For example:

```
#include <stddef.h>
wchar_t a[6] = L"Hello";
```

All char objects are stored in 8 bits. All wchar_t objects are stored as unsigned int objects in 32 bits. The value of a given character is determined by the character set being used. In this text, the ASCII character set is used in all examples. See Appendix C for a complete list of ASCII equivalents, in decimal, octal, and hexadecimal radixes.

To aid portability, declare char objects that will be used in arithmetic as signed char or unsigned char. For example:

```
signed char letter;
unsigned char symbol_1, symbol_2;
signed char alpha = 'A';  /* alpha is declared and initialized as 'A' */
```

Strings are arrays of characters terminated by the null character (\0). Section 1.9.3 has more information on the syntactic rules of using strings; Chapter 4 has information on declaring string literals.

3.3 Floating-Point Types

The floating-point types are:

- float—32 bits
- double—64 bits
- long double *(OpenVMS Alpha)*—128 bits by default, with the option for 64 bits
- long double *(Tru64 UNIX)*—64 bits in current versions of Tru64 UNIX
- long double *(VAX)*—64 bits
- float _Complex *(Alpha, I64)*
- double _Complex *(Alpha, I64)*
- long double _Complex *(Alpha, I64)*

Use the floating-point types for variables, constants, and function return values with fractional parts, or where the value exceeds the storage range available with the integral types. The following examples show sample floating-point type declarations (and initializations):

```
float x = 35.69;
double y = .0001;
double z = 77.0e+10;
float Q = 99.9e+99;                /*  Exceeds allowable range   */
```

3.3.1 Complex Type *(Alpha, I64)*

The C99 standard introduces a built-in complex data type similar to the Fortran type, in all three precisions (`float _Complex`, `double _Complex`, and `long double _Complex`). It also has an associated header file, `<complex.h>` The `<complex.h>` header file defines a macro spelled "complex", intended to be the preferred way to refer to the types. (See Section 9.2).

A complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; the first element is equal to the real part, and the second element to the imaginary part, of the complex number.

The type is similar to the Fortran type in its use. There is no special syntax for constants; instead there is a new keyword `_Complex_I`, which has a complex value whose real part is zero and whose imaginary part is 1.0. The header file defines a macro `I` that expands to `_Complex_I`, and so a complex constant with equal real and imaginary parts of 2.0 would be written as `2.0 + 2.0*I`.

There are some known issues with complex types on HP C:

- The complex data types are not available when using the /FLOAT=D_ FLOAT command-line option. This is a permanent restriction.
- The complex types and functions other than `long double complex` are available on versions of OpenVMS prior to Version 7.3. The `long double complex` type is available with OpenVMS Version 7.3.
- Initialized declarations of `long double complex` variables cause a compiler assertion failure when generating a machine-code listing.
- Functions named `cabs`, `cabsf`, and `cabsl` have traditionally been declared in `<math.h>` using a struct representation to hold two floating values. This is not compatible with the calling standard for passing complex values. To access working cabs functions, you must include `<complex.h>` before you include `<math.h>`.

3.3.2 Imaginary Type *(Alpha, I64)*

The C99 standard reserves the keyword `_Imaginary` for use as a type-specifier in conjunction with an experimental/optional feature called a "pure imaginary" type, specified in informative Annex G. In HP C, use of the `_Imaginary` keyword produces a warning, which is resolved by treating it as an ordinary identifier.

3.4 Derived Types

There are five derived types in C:

- Function types
- Pointer types
- Array types
- Structure types
- Union types

The following sections describe these derived types.

A derived type is formed by using one or more basic types in combination. Using derived types, an infinite variety of new types can be formed. The array and structure types are collectively called the *aggregate* types. Note that the aggregate types do not include union types, but a union may contain an aggregate member.

3.4.1 Function Type

A function type describes a function that returns a value of a specified type. If the function returns no value, it should be declared as "function returning `void`" as follows:

```
void function1 ();
```

In the following example, the data type for the function is "function returning `int`":

```
int uppercase(int lc)
{
  int uc = lc + 0X20;
  return uc;
}
```

Chapter 4 discusses declarations in general. Chapter 5 covers functions specifically, including their declarations, parameters, and argument passing.

3.4.2 Pointer Type

A pointer type describes a value that represents the address of an object of a stated type. A pointer is stored as an integral value that references the address of the target object. Pointer types are derived from other types, called the *referenced type* of the pointer. For example:

```
int *p;          /*  p is a pointer to an int type                */
double *q();     /*  q is a function returning a pointer to an
                     object of type double                        */
int (*r)[5];     /*  r is a pointer to an array of five elements  */
                 /*  (r holds the address to the first element of
                     the array)                                   */
const char s[6]; /*  s is a const-qualified array of 6 elements   */
```

The pointer itself can have any storage class, but the object addressed by the pointer cannot have the `register` storage class or be a bit field. Pointers to qualified or unqualified versions of compatible types have the same representation and alignment requirements as the target type. Pointers to other types need not have the same representation or alignment requirements.

The construction `void *` designates a generic ‘‘pointer to `void`’’ type. The `void *` construction can be used to point to an object of any type, and it is most useful when a pointer is needed to point to the address of objects with different or unknown types (such as in a function prototype). A pointer to `void` can also be converted to or from a pointer of any other type, and has the same representation and alignment requirements as a pointer to a character type.

A pointer to the address 0 (zero) is called a *null pointer*. Null pointers are often used to indicate that no more members of a list exist (for example, when using pointers to show the next member of the list). Dereferencing a null pointer with the * or subscripting operators leads to unpredictable and usually very unfavorable results.

See Chapter 4 for details on the syntax of pointer declarations.

3.4.3 Array Type

An array type can be formed from any valid completed type. Completion of an array type requires that the number and type of array members be explicitly or implicitly specified. The member types can be completed in the same or a different compilation unit. Arrays cannot be of `void` or function type, since the `void` type cannot be completed and function types are not object types requiring storage.

Typically, arrays are used to perform operations on some homogeneous set of values. The size of the array type is determined by the data type of the array and the number of elements in the array. Each element in an array has the same type. For example, the following definition creates an array of four characters:

```
char x[] = "Hi!"   /*  Declaring an array x   */;
```

Each of the elements has the size of a `char` object, 8 bits. The size of the array is determined by its initialization; in the previous example, the array has three explicit elements plus one null character. Four elements of 8 bits each results in an array with a size of 32 bits.

An array is allocated contiguously in memory, and cannot be empty (that is, have no members). An array can have only one dimension. To create an array of ''two dimensions,'' declare an array of arrays, and so on.

It is possible to declare an array of unknown size; this sort of declaration is called an *incomplete array declaration*, because the size is not specified. The following example shows an incomplete declaration:

```
int x[];
```

The size of an array declared in this manner must be specified elsewhere in the program. (See Section 4.7 for more information on declaring incomplete arrays and initializing arrays.)

Character strings (string literals) are stored in the form of an array of `char` or `wchar_t` type, and are terminated by the null character (\0).

An array in C has only one dimension. An array of arrays can be declared, however, to create a *multidimensional array*. The elements of these arrays are stored in increasing addresses so that the rightmost subscript varies most rapidly. This is called *row-major order*, and is analogous to a car's odometer. For example, in an array of two arrays declared as `int a[2][3];` the elements are stored in this order:

```
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]
```

3.4.4 Structure Type

A structure type is a sequentially allocated nonempty set of objects, called *members*. Structures let you group heterogeneous data. They are much like records in Pascal. Unlike arrays, the elements of a structure need not be of the same data type. Also, elements of a structure are accessed by name, not by subscript. The following example declares a structure `employee`, with two structure variables (`ed` and `mary`) of the structure type `employee`:

```
struct employee { char name[30]; int age; int empnumber; };
struct employee ed, mary;
```

Structure members can have any type except an incomplete type, such as the `void` type or a function type. Structures can contain pointers to objects of their own type, but they cannot contain an object of their own type as a member; such an object would have an incomplete type. For example:

```
/*  This is invalid. */
struct employee {
  char name[30];
  struct employee div1;       /*  This is invalid. */

  int *f();                   /*  This is also invalid. */

};
```

The following example, however, is valid:

```
struct employee {
  char name[30];
  struct employee *div1;/*  Member can contain pointer to employee
                            structure.                              */
  int (*f)();           /*  Pointer to a function returning int     */
};
```

The name of a declared structure member must be unique within the structure, but it can be used in another nested or unnested structure or name spaces to refer to a different object. For example:

```
struct {
  int a;
  struct {
    int a;  /* This 'a' refers to a different object
               than the previous 'a'                */

  } nested;

};
```

As an extension, the relaxed modes of HP C allow a strucure or union to declare nested structure or union members without specifying names for the members - called *anonymous* members. The effect is as if the names of the members of the anonymous nested structure or union were declared directly within the containing structure or union, rather than being nested. Therefore, in the preceding example, if the identifier `nested` were omitted from the inner `struct` declaration, there would be an error because the member `a` in the inner structure would conflict with the member `a` in the containing structure. This is similar to the anonymous union feature of the C++ language (except that it is permitted for both structures and unions), and to the `variant_struct` and `variant_union` features unique to VAX C.

Chapter 4 contains more examples on structures and their declarations.

The compiler assigns storage for structure members in the order of member declaration, with increasing memory addresses for subsequent members. The first member always begins at the starting address of the structure itself. Subsequent members are aligned per the *alignment unit*, which may differ depending on the member sizes in the structure. A structure may contain padding (unused bits) so that members of an array of such structures are properly aligned, and the size of the structure is the amount of storage necessary for all members plus any padded space needed to meet alignment requirements. See your system's HP C documentation for platform-specific information about structure alignment and representation.

A pragma is available to change the alignment of a structure on one platform to match that of structures on other platforms. See Section B.29 for more information on this pragma.

3.4.5 Union Type

A union type can store objects of different types at the same location in memory. The different union members can occupy the same location at different times in the program. The declaration of a union includes all members of the union, and lists the possible object types the union can hold. The union can hold any one member at a time—subsequent assignments of other members to the union overwrite the existing object in the same storage area.

Unions can be named with any valid identifier. An empty union cannot be declared, nor can a union contain an instance of itself. A member of a union cannot have a `void`, function, or incomplete type. Unions can contain pointers to unions of their type.

Another way to look at a union is as a single object that can represent objects of different types at different times. Unions let you use objects whose type and size can change as the program progresses, without using machine-dependent constructions. Some other languages call this concept a *variant record*.

The syntax for defining unions is very similar to that for structures. Each union type definition creates a unique type. Names of union members must be unique within the union, but they can be duplicated in other nested or unnested unions or name spaces. For example:

```
union {
  int a;

  union foo {

    int a;  /* This 'a' refers to a different object
               than the previous 'a'                */

  } nested;

};
```

Note that as an extension, relaxed modes of HP C permit anonymous union members as in the C++ language.

The size of a union is the amount of storage necessary for its largest member, plus any padding needed to meet alignment requirements.

Once a union is defined, a value can be assigned to any of the objects declared in the union declaration. For example:

```
union name {
  double dvalue;
  struct x { int value1; int value2; };
  float fvalue;
} alberta;
alberta.dvalue = 3.141596; /* Assigns the value of pi to the union object */
```

Here, `alberta` can hold a `double`, `struct`, or `float` value. The programmer has responsibility for tracking the current type of object contained in the union. An assignment expression can be used to change the type of value held in the union.

Undefined behavior results when a union is used to store a value of one type, and then the value is accessed through another type. For example:

```
/*
    Assume that 'node' is a typedef_name for objects for which
    information has been entered into a hash table;

    'hash_entry' is a structure describing an entry in the hash table.
    The member 'hash_value' is a pointer to the relevant 'node'.
 */
typedef struct hash_entry
{
   struct hash_entry *next_hash_entry;
   node   *hash_value;
   /* ... other information may be present ... */
} hash_entry;

extern hash_entry *hash_table [512];
```

```
/*
    'hash_pointer' is a union whose members are a pointer to a
    'node' and a structure containing three bit fields that
    overlay the pointer value.  Only the second bit field is
    being used, to extract a value from the middle
    of the pointer to be used as an index into the hash table.
    Note that nine bits gives a range of values from 0 to 511;
    hence, the size of 'hash_table' above.
 */
typedef union
{
   node *node_pointer;
   struct
   {
    unsigned : 4;
    unsigned  index : 9;
    unsigned :19;
   } bits;
} hash_pointer;
```

3.5 void Type

The `void` type is an incomplete type that cannot be completed.

The `void` type has three important uses:

- To signify that a function returns no value
- To indicate a generic pointer (one that can point to any type object)
- To specify a function prototype with no arguments

The following example shows how `void` is used to define a function, with no parameters, that does not return a value:

```
void message(void)
{
  printf ("Stop making sense!");
}
```

The next example shows a function prototype for a function that accepts a pointer to any object as its first and second argument:

```
void memcopy (void *dest, void *source, int length);
```

A pointer to the `void` type has the same representation and alignment requirements as a pointer to a character type. The `void *` type is a derived type based on `void`.

The void type can also be used in a cast expression to explicitly discard or ignore a value. For example:

```
int tree(void);

void main()
{
  int i;

  for (; ; (void)tree()){...}  /* void cast is valid                  */

  for (; (void)tree(); ;){...} /* void cast is NOT valid, because the */
                               /* value of the second expression in a */
                               /* for statement is used               */

  for ((void)tree(); ;) {...}  /* void cast is valid                  */

}
```

A void expression has no value, and cannot be used in any context where a value is required.

3.6 Enumerated Types

An enumerated type is used to specify the possible values of an object from a predefined list. Elements of the list are called *enumeration constants*. The main use of enumerated types is to explicitly show the symbolic names, and therefore the intended purpose, of objects whose values can be represented with integer values.

Objects of enumerated type are interpreted as objects of type signed int, and are compatible with objects of other integral types.

Note

In RELAXED mode, the compiler allows enumeration constants to have type other than signed int.

The compiler automatically assigns integer values to each of the enumeration constants, beginning with 0. The following example declares an enumerated object background_color with a list of enumeration constants:

```
enum colors { black, red, blue, green, white } background_color;
```

Later in the program, a value can be assigned to the object background_color:

```
background_color = white;
```

In this example, the compiler automatically assigns the integer values as follows: `black` = 0, `red` = 1, `blue` = 2, `green` = 3, and `white` = 4. Alternatively, explicit values can be assigned during the enumerated type definition:

```
enum colors { black = 5, red = 10, blue, green = 7, white = green+2 };
```

Here, `black` equals the integer value 5, `red` = 10, `blue` = 11, `green` = 7, and `white` = 9. Note that `blue` equals the value of the previous constant (`red`) plus one, and `green` is allowed to be out of sequential order.

Because the ANSI C standard is not strict about assignment to enumerated types, any assigned value not in the predefined list is accepted without complaint.

3.7 Type Qualifiers

There are four type qualifiers:

- `const`
- `volatile`
- `__unaligned` (axp)
- `__restrict` (pointer type only)

Type qualifiers were introduced by the ANSI C standard to, in part, give you greater control over the compiler's optimizations. The `const` and `volatile` type qualifiers can be applied to any type. The `__restrict` type qualifier can be applied only to pointer types.

Note that because the `__restrict` type qualifier is not part of the 1989 ANSI C standard, this keyword has double leading underscores. The next version (9X) of the C standard is expected to adopt the keyword `restrict` with the same semantics described in this section.

The use of `const` gives you a method of controlling write access to an object, and eliminates potential side effects across function calls involving that object. This is because a side effect is an alteration of an object's storage and `const` prohibits such alteration.

Use `volatile` to qualify an object that can be changed by other processes or hardware. The use of `volatile` disables optimizations with respect to referencing the object. If an object is volatile qualified, it may be changed between the time it is initialized and any subsequent assignments. Therefore, it cannot be optimized.

Function parameters, however, do not all share the type qualification of one parameter. For example:

```
int f( const int a, int b)   /*  a is const qualified; b is not  */
```

When using a type qualifier with an array identifier, the elements of the array are qualified, not the array type itself.

The following declarations and expressions show the behavior when type qualifiers modify an array or structure type:

```
const struct s { int mem; } cs = { 1 };
struct s ncs;                         /* ncs is modifiable         */
typedef int A[2][3];
const A a = {{4, 5, 6}, {7, 8, 9}};  /*  array of array of const  */
                                      /*  int's                    */
int *pi;
const int *pci;

ncs = cs;            /*  Valid                                     */
cs = ncs;            /*  Invalid, cs is const-qualified            */
pi = &ncs.mem;       /*  Valid                                     */
pi = &cs.mem;        /*  Violates type constraints for = operator  */
pci = &cs.mem;       /*  Valid                                     */
pi = a[0];           /*  Invalid; a[0] has type "const int *"      */
```

3.7.1 const Type Qualifier

Use the `const` type qualifier to qualify an object whose value cannot be changed. Objects qualified by the `const` keyword cannot be modified. This means that an object declared as `const` cannot serve as the operand in an operation that changes its value; for example, the ++ and −− operators are not allowed on objects qualified with `const`. Using the `const` qualifier on an object protects it from the side effects caused by operations that alter storage.

The declaration of `const`-qualified objects can be slightly more complicated than that for nonqualified types. Here are some examples, with explanatory comments:

```
const int x = 44;   /*  const qualification of int type.
                        The value of x cannot be modified. */
const int *z;       /*  Pointer to a constant integer.
                        The value in the location pointed
                        to by z cannot be modified.        */
int * const ptr;    /*  A constant pointer, a pointer
                        that will always point to the
                        same location                      */
const int *const p; /*  A constant pointer to a constant
                        integer: neither the pointer or
                        the integer can be modified.       */
const const int y;  /*  Illegal - redundant use of const   */
```

The following rules apply to the const type qualifier:

- The const qualifier can be used to qualify any data type, including a single member of a structure or union.
- If const is specified when declaring an aggregate type, all members of the aggregate type are treated as objects qualified with const. When const is used to qualify a member of an aggregate type, only that member is qualified. For example:

```
const struct employee {
    char *name;
    int   birthdate; /* name, birthdate, job_code, and salary are */
    int   job_code;  /* treated as though declared with const.    */
    float salary;
    } a, b;          /* All members of a and b are const-qualified*/
struct employee2 {
    char *name;
    const int birthdate;  /*  Only this member is qualified    */
    int job_code;
    float salary;
    } c, d;
```

  All members in the previous structure are qualified with const. If the tag employee is used to specify another structure later in the program, the const qualifier does not apply to the new structure's members unless explicitly specified.

- The const qualifier can be specified with the volatile qualifier. This is useful, for example, in a declaration of a data object that is immutable by the source process but can be changed by other processes, or as a model of a memory-mapped input port such as a real-time clock.
- The address of a non-const object can be assigned to a pointer to a const object (with an explicit const specifier), but that pointer cannot be used to alter the value of the object. For example:

```
const int i = 0;
int j = 1;
const int *p = &i; /* Explicit const specifier required      */
int *q = &j;
*p = 1;            /* Error -- attempt to modify a const-
                      qualified object through a pointer      */
*q = 1;            /* OK                                      */
```

- Attempting to modify a const object using a pointer to a non-const qualified type causes unpredictable behavior.

3.7.2 volatile Type Qualifier

Any object whose type includes the `volatile` type qualifier indicates that the object should not be subject to compiler optimizations altering references to, or modifications of, the object.

Note

`volatile` objects are especially prone to side effects. (See Section 2.5.)

Optimizations that are defeated by using the `volatile` specifier can be categorized as follows:

- Optimizations that alter an object's duration; for example, cases where references to the object are shifted or moved to another part of the program.
- Optimizations that alter an object's locality; for example, cases where a variable serving as a loop counter is stored in a register to save the cost of doing a memory reference.
- Optimizations that alter an object's existence; for example, loop induction to actually eliminate a variable reference.

An object without the `volatile` specifier does not compel the compiler to perform these optimizations; it indicates that the compiler has the freedom to apply the optimizations depending on program context and compiler optimization level.

The `volatile` qualifier forces the compiler to allocate memory for the `volatile` object, and to always access the object from memory. This qualifier is often used to declare that an object can be accessed in some way not under the compiler's control. Therefore, an object qualified by the `volatile` keyword can be modified or accessed in ways by other processes or hardware, and is especially vulnerable to side effects.

The following rules apply to the use of the `volatile` qualifier:

- The `volatile` qualifier can be used to qualify any data type, including a single member of a structure or union.
- Redundant use of the `volatile` keyword elicits a warning message. For example:

```
volatile volatile int x;
```

- When volatile is used with an aggregate type declaration, all members of the aggregate type are qualified with volatile. When volatile is used to qualify a member of an aggregate type, only that member is qualified. For example:

```
volatile struct employee {
    char *name;
    int   birthdate; /* name, birthdate, job_code, and salary are */
    int   job_code;  /* treated as though declared with volatile. */
    float salary;
    } a,b;          /*  All members of a and b are volatile-qualified  */
struct employee2 {
    char *name;
    volatile int birthdate;  /*  Only this member is qualified    */
    int job_code;
    float salary;
    } c, d;
```

  If the tag employee is used to specify another structure later in the program, the volatile qualifier does not apply to the new structure's members unless explicitly specified.

- The const qualifier can be used with the volatile qualifier. This is useful, for example, in a declaration of a data object that is immutable by the source process but can be changed by other processes, or as a model of a memory-mapped input port such as a real-time clock.

- The address of a non-volatile object can be assigned to a pointer that points to a volatile object. For example:

```
const int *intptr;
volatile int x;
intptr = &x;
```

  Likewise, the address of a volatile object can be assigned to a pointer that points to a non-volatile object.

3.7.3 __unaligned Type Qualifier

Use this data-type qualifier in pointer definitions to indicate to the compiler that the data pointed to is not properly aligned on a correct address. (To be properly aligned, the address of an object must be a multiple of the size of the type. For example, two-byte objects must be aligned on even addresses.)

When data is accessed through a pointer declared __unaligned, the compiler generates the additional code necessary to copy or store the data without causing alignment errors. It is best to avoid use of misaligned data altogether, but in some cases the usage may be justified by the need to access packed structures, or by other considerations.

Here is an example of a typical use of __unaligned:

```
typedef enum {int_kind, float_kind, double_kind} kind;
void foo(void *ptr, kind k) {
    switch (k) {
    case int_kind:
        printf("%d", *(__unaligned int *)ptr);
        break;
    case float_kind:
        printf("%f", *(__unaligned float *)ptr);
        break;
    case double_kind:
        printf("%f", *(__unaligned double *)ptr);
        break;
    }
}
```

3.7.4 __restrict Type Qualifier

Use the __restrict type qualifier on the declaration of a pointer type to indicate that the pointer is subject to compiler optimizations. Restricted pointers are expected to be an addition to the 9X revision of the ISO C Standard. Using restricted pointers judiciously can often improve the quality of code output by the compiler.

3.7.4.1 Rationale

The following sections describe the rationale for restricted-pointer support.

3.7.4.1.1 Aliasing For many compiler optimizations, ranging from simply holding a value in a register to the parallel execution of a loop, it is necessary to determine whether two distinct lvalues designate distinct objects. If the objects are not distinct, the lvalues are said to be *aliases*. If the compiler cannot determine whether or not two lvalues are aliases, it must assume that they are aliases and suppresses various optimizations.

Aliasing through pointers presents the greatest difficulty, because there is often not enough information available within a single function, or even within a single compilation unit, to determine whether two pointers can point to the same object. Even when enough information is available, this analysis can require substantial time and space. For example, it could require an analysis of a whole program to determine the possible values of a pointer that is a function parameter.

3.7.4.1.2 Library Examples Consider how potential aliasing enters into implementations in C of two Standard C library functions memmove and memcpy:

- There are no restrictions on the use of memmove, and the sample implementation that follows adheres to the model described in the revised ISO C Standard by copying through a temporary array.
- Because memcpy cannot be used for copying between overlapping arrays, its implementation can be a direct copy.

The following example contrasts sample implementations of the memcpy and memmove functions:

```
/* Sample implementation of memmove */

   void *memmove(void *s1, const void *s2, size_t n) {
           char * t1 = s1;
           const char * t2 = s2;
           char * t3 = malloc(n);
           size_t i;
           for(i=0; i<n; i++) t3[i] = t2[i];
           for(i=0; i<n; i++) t1[i] = t3[i];
           free(t3);
           return s1;
   }

/* Sample implementation of memcpy */

   void *memcpy(void *s1, const void *s2, size_t n);
           char * t1 = s1;
           const char * t2 = s2;
           while(n -> 0) *t1++ = *t2++;
           return s1;
   }
```

The restriction on memcpy is expressed only in its description in the Standard, and cannot be expressed directly in its implementation in C. While this allows the source-level optimization of eliminating the temporary used in memmove, it does not provide for compiler optimization of the resulting single loop.

In many architectures, it is faster to copy bytes in blocks, rather than one at a time:

- The implementation of memmove uses malloc to obtain the temporary array, and this guarantees that the temporary is disjoint from the source and target arrays. From this, a compiler can deduce that block copies can safely be used for both loops (if the compiler recognizes malloc as a special function that allocates new memory).

- The implementation of memcpy, on the other hand, provides no basis for the compiler to rule out the possibility that, for example, s1 and s2 point to successive bytes. Therefore, unconditional use of block copies does not appear to be safe, and the code generated for the single loop in memcpy might not be as fast as the code for each loop in memmove.

3.7.4.1.3 Overlapping Objects The restriction in the description of memcpy in the Standard prohibits copying between *overlapping objects*. An *object* is a region of data storage, and except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.

Consider the following example:

```
/* memcpy between rows of a matrix */

void f1(void) {
        extern char a[2][N];
        memcpy(a[1], a[0], N);
}
```

In this example:

- The objects are exactly the regions of data storage pointed to by the pointers and dynamically determined to be of N bytes in length (that is, treated as an array of N elements of character type).
- The objects are not the largest objects into which the arguments can be construed as pointing.
- The call to memcpy has defined behavior.
- The behavior is defined because the pointers point into different (non-overlapping) objects.

Now consider the following example:

```
/* memcpy between halves of an array */

void f2(void) {
        extern char b[2*N];
        memcpy(b+N, b, N);
}
```

In this example:

- Objects are defined as regions of data storage unrelated to declarations or types.
- For memcpy, a contiguous sequence of elements within an array can be regarded as an object in its own right.

- The objects are not the smallest contiguous sequence of bytes that can be construed; they are exactly the regions of data storage starting at the pointers and of N bytes in length.
- The non-overlapping halves of array `b` can be regarded as objects in their own rights.
- Behavior is defined.

The length of an object is determined by various methods:

- For strings in which all elements are accessed, length is inferred by null-byte termination.
- For `mbstowcs`, `wcstombs`, `strftime`, `vsprintf`, `sscanf`, `sprintf`, and all other similar functions, objects and lengths are dynamically determined.

3.7.4.1.4 Restricted Pointer Prototype for memcpy If an aliasing restriction like the one for `memcpy` could be expressed in a function definition, then it would be available to a compiler to facilitate effective pointer alias analysis. The `__restrict` type qualifier accomplishes this by specifying in the declaration of a pointer that the pointer provides *exclusive initial access* to the object to which it points, as though the pointer were initialized with a call to `malloc`.

The following prototype for `memcpy` both expresses the desired restriction and is compatible with the current prototype:

```
void *memcpy(void * ___restrict s1, const void * ___restrict s2, size_t n);
```

3.7.4.2 Formal Definition of the __restrict Type Qualifier

The following definition of restricted pointers supports expression of aliasing restrictions in as many paradigms as possible. This is helpful in converting existing programs to use restricted pointers, and allows more freedom of style in new programs.

This definition, therefore, allows restricted pointers to be:

- Modifiable
- Members of structures and elements of arrays
- Strongly scoped, in the sense that a restricted pointer declared in a nested block makes a non-aliasing assertion only within that block

Definition

A pointer is designated as a restricted pointer by specifying the `__restrict` type qualifier on its declaration.

The formal definition of a restricted pointer as proposed for inclusion in the revised ISO C Standard follows:

> Let D be a declaration of an ordinary identifier that provides a means of designating an object P as a restrict-qualified pointer.
>
> If D appears inside a block and does not have storage-class extern, let B denote the block. If D appears in the list of parameter declarations of a function definition, let B denote the associated block. Otherwise, let B denote the block of main (or the block of whatever function is called at program startup, in a freestanding environment).
>
> In what follows, a pointer expression E is said to be based on object P if (at some sequence point in the execution of B prior to the evaluation of E) modifying P to point to a copy of the array object into which it formerly pointed would change the value of E. (In other words, E depends on the value of P itself rather than on the value of an object referenced indirectly through P. For example, if identifier `p` has type `(int ** restrict)`, then the pointer expressions `p` and `p+1` are based on the restricted pointer object designated by `p`, but the pointer expressions `*p` and `p[1]` are not.)
>
> During each execution of B, let O be the array object that is determined dynamically by all references through pointer expressions based on P. All references to values of O shall be through pointer expressions based on P. Furthermore, if P is assigned the value of a pointer expression E that is based on another restricted pointer object P2, associated with block B2, then either the execution of B2 shall begin before the execution of B, or the execution of B2 shall end prior to the assignment. If this requirement is not met, then the behavior is undefined.
>
> Here an execution of B means that portion of the execution of the program during which storage is guaranteed to be reserved for an instance of an object that is associated with B and has automatic storage duration. A reference to a value means either an access to or a modification of the value. During an execution of B, attention is confined to those references that are actually evaluated (this excludes references that appear in unevaluated expressions, and also excludes references that are

"available," in the sense of employing visible identifiers, but do not actually appear in the text of B).

A translator is free to ignore any or all aliasing implications of uses of restrict.

3.7.4.3 Examples

The formal definition of the `__restrict` type qualifier can be difficult to grasp, but simplified explanations tend to be less accurate and complete. The essence of the definition is that the `__restrict` type qualifier is an assertion by the programmer that whenever a memory access is made through a restricted pointer, the only aliases the compiler need consider are other accesses made through the same pointer.

Much of the complexity is in defining exactly what is meant for an access to be made through a pointer (the *based-on* rules), and specifying how a restricted pointer can be assigned the value of another restricted pointer, while limiting the aliasing potential to occur only at block boundaries. Examples can be the best way to understand restricted pointers.

The following examples show the use of restricted pointers in various contexts.

3.7.4.3.1 File Scope Restricted Pointers A file scope restricted pointer is subject to very strong restrictions. It should point into a single array object for the duration of the program. That array object must not be referenced both through the restricted pointer and through either its declared name (if it has one) or another restricted pointer.

Because of these restrictions, references through the pointer can be optimized as effectively as references to a static array through its declared name. File scope restricted pointers are therefore useful in providing access to dynamically allocated global arrays.

In the following example, a compiler can deduce from the `__restrict` type qualifiers that there is no potential aliasing among the names `a`, `b`, and `c`:

```
/* File Scope Restricted Pointer */

float * __restrict a, * __restrict b;
float c[100];

int init(int n) {
    float * t = malloc(2*n*sizeof(float));
    a = t;       /* a refers to 1st half. */
    b = t + n;   /* b refers to 2nd half. */
   }
```

Notice how the single block of allocated storage is subdivided into two unique arrays in the function `init`.

3.7.4.3.2 Function Parameters Restricted pointers are also very useful as pointer parameters of a function. Consider the following example:

```
/* Restricted pointer function parameters */

float x[100];
float *c;

void f3(int n, float * __restrict a, float * const b) {
    int i;
    for ( i=0; i<n; i++ )
        a[i] = b[i] + c[i];
}
void g3(void) {
    float d[100], e[100];
    c = x; f3(100,   d,    e); /* Behavior defined.   */
           f3( 50,   d, d+50); /* Behavior defined.   */
           f3( 99, d+1,    d); /* Behavior undefined. */
    c = d; f3( 99, d+1,    e); /* Behavior undefined. */
           f3( 99,   e,  d+1); /* Behavior defined.   */
}
```

In the function `f3`, it is possible for a compiler to infer that there is no aliasing of modified objects, and so to optimize the loop aggressively. Upon entry to `f3`, the restricted pointer `a` must provide exclusive access to its associated array. In particular, within `f3` neither `b` nor `c` may point into the array associated with `a`, because neither is assigned a pointer value based on `a`. For `b`, this is evident from the `const` qualifier in its declaration, but for `c`, an inspection of the body of `f3` is required.

Two of the calls shown in `g3` result in aliasing that is inconsistent with the `__restrict` qualifier, and their behavior is undefined. Note that it is permitted for `c` to point into the array associated with `b`. Note also that, for these purposes, the "array" associated with a particular pointer means only that portion of an array object that is actually referenced through that pointer.

3.7.4.3.3 Block Scope A block-scope restricted pointer makes an aliasing assertion that is limited to its block. This is more natural than allowing the assertion to have function scope. It allows local assertions that apply only to key loops, for example. It also allows equivalent assertions to be made when inlining a function by converting it into a macro.

In the following example, the original restricted-pointer parameter is represented by a block-scope restricted pointer:

```
/*  Macro version of f3 */

float x[100];
float *c;

#define f3(N, A, B)                                     \
{   int n = (N);                                        \
    float * ___restrict a = (A);                           \
    float * const    b = (B);                           \
    int i;                                              \
    for ( i=0; i<n; i++ )                               \
        a[i] = b[i] + c[i];                             \
}
```

3.7.4.3.4 Members of Structures A restricted-pointer member of a structure makes an aliasing assertion. The scope of that assertion is the scope of the ordinary identifier used to access the structure.

Therefore, although the structure type is declared at file scope in the following example, the assertions made by the declarations of the parameters of `f4` have block (of the function) scope.

```
/* Restricted pointers as members of a structure */

struct t {     /* Restricted pointers assert that    */
    int n;     /* members point to disjoint storage. */
    float * ___restrict p;
    float * ___restrict q;
};

void f4(struct t r, struct t s) {
    /* r.p, r.q, s.p, s.q should all point to    */
    /* disjoint storage during each execution of f4. */
    /* ... */
}
```

3.7.4.3.5 Type Definitions A `__restrict` qualifier in a `typedef` makes an aliasing assertion when the `typedef` name is used in the declaration of an ordinary identifier that provides access to an object. As with members of structures, the scope of the latter identifier, not the scope of the `typedef` name, determines the scope of the aliasing assertion.

3.7.4.3.6 Expressions Based on Restricted Pointers Consider the following example:

```
/* Pointer expressions based on p */

#include <stdlib.h>
#include <string.h>

struct t { int * q; int i; } a[2] = { /* ... */ };

void f5(struct t * __restrict p, int c)
{
     struct t * q;
     int n;
     if(c) {
         struct t * r;
         r = malloc(2*sizeof(*p));
         memcpy(r, p, 2*sizeof(*p));
         p = r;
     }
     q = p;
     n = (int)p;

     /* - - - - - - - - - - - - - - - - - - - - - - - - -

       Pointer expressions     Pointer expressions
       based on p:             not based on p:
       -------------------     -------------------
       p                       p-&gt;q
       p+1                     p[1].q
       &amp;p[1]                   &amp;p
       &amp;p[1].i
       q                       q-&gt;p
       ++q
       (char *)p               (char *)(p-&gt;i)
       (struct t *)n           ((struct t *)n)->q
     - - - - - - - - - - - - - - - - - - - - - - - - - */
}

main() {
    f5(a, 0);
    f5(a, 1);
}
```

In this example, the restricted pointer parameter `p` is potentially adjusted to point into a copy of its original array of two structures. By definition, a subsequent pointer expression is said to be based on `p` if and only if its value is changed by this adjustment.

In the comment:

- The values of the pointer expressions in the first column are changed by this adjustment, and so those expressions are based on `p`.

- The values of the pointer expressions in the second column are not changed by the adjustment, and so those expressions are not based on `p`.

This can be verified by adding appropriate print statements for the expressions and comparing the values produced by the two calls of `f5` in `main`.

Notice that the definition of "based on" applies to expressions that rely on implementation-defined behavior. This is illustrated in the example, which assumes that the casts `(int)` followed by `(struct t *)` give the original value.

3.7.4.3.7 Assignments Between Restricted Pointers Consider one restricted pointer "newer" than another if the block with which the first is associated begins execution after the block associated with the second. Then the formal definition allows a newer restricted pointer to be assigned a value based on an older restricted pointer. This allows, for example, a function with a restricted-pointer parameter to be called with an argument that is a restricted pointer.

Conversely, an older restricted pointer can be assigned a value based on a newer restricted pointer only after execution of the block associated with the newer restricted pointer has ended. This allows, for example, a function to return the value of a restricted pointer that is local to the function, and the return value then to be assigned to another restricted pointer.

The behavior of a program is undefined if it contains an assignment between two restricted pointers that does not fall into one of these two categories. Some examples follow:

```
/* Assignments between restricted pointers */

int * __restrict p1, * __restrict p2;

void f6(int * __restrict q1, * __restrict q2)
{
    q1 = p1;     /* Valid behavior     */
    p1 = p2;     /* Behavior undefined */
    p1 = q1;     /* Behavior undefined */
    q1 = q2;     /* Behavior undefined */
    {
        int * __restrict r1, * __restrict r2;
        ...
        r1 = p1; /* Valid behavior     */
        r1 = q1; /* Valid behavior     */
        r1 = r2; /* Behavior undefined */
        q1 = r1; /* Behavior undefined */
        p1 = r1; /* Behavior undefined */
        ...
    }
}
```

3.7.4.3.8 Assignments to Unrestricted Pointers

The value of a restricted pointer can be assigned to an unrestricted pointer, as in the following example:

```
/* Assignments to unrestricted pointers */

void f7(int n, float * __restrict r, float * __restrict s) {
    float * p = r, * q = s;
    while(n->0)
        *p++ = *q++;
}
```

The HP C compiler tracks pointer values and optimizes the loop as effectively as if the restricted pointers `r` and `s` were used directly, because in this case it is easy to determine that `p` is based on `r`, and `q` is based on `s`.

More complicated ways of combining restricted and unrestricted pointers are unlikely to be effective because they are too difficult for a compiler to analyze. As a programmer concerned about performance, you must adapt your style to the capabilities of the compiler. A conservative approach would be to avoid using both restricted and unrestricted pointers in the same function.

3.7.4.3.9 Ineffective Uses of Type Qualifiers

Except where specifically noted in the formal definition, the `__restrict` qualifier behaves in the same way as `const` and `volatile`.

In particular, it is not a constraint violation for a function return type or the *type-name* in a cast to be qualified, but the qualifier has no effect because function call expressions and cast expressions are not lvalues.

Thus, the presence of the `__restrict` qualifier in the declaration of `f8` in the following example makes no assertion about aliasing in functions that call `f8`:

```
/* Qualified function return type and casts */

float * __restrict f8(void)  /* No assertion about aliasing. */
{
    extern int i, *p, *q, *r;

    r = (int * __restrict)q; /* No assertion about aliasing. */

    for(i=0; i<100; i++)
        *(int * __restrict)p++ =   r[i];  /* No assertion    */
                                      /* about aliasing. */
    return p;
}
```

Similarly, the two casts make no assertion about aliasing of the references through the pointers `p` and `r`.

3.7.4.3.10 Constraint Violations It is a constraint violation to restrict-qualify an object type that is not a pointer type, or to restrict-qualify a pointer to a function:

```
/*___restrict cannot qualify non-pointer object types: */

int ___restrict x;    /* Constraint violation */
int ___restrict *p;   /* Constraint violation */

/* ___restrict cannot qualify pointers to functions: */

float (* ___restrict f9)(void); /* Constraint violation */
```

3.8 Type Definition

The keyword `typedef` is used to define a type synonym. In such a definition, the identifiers name *types* instead of *objects*. One such use is to define an abbreviated name for a lengthy or confusing type definition.

A type definition does not create a new basic data type; it creates an alias for a basic or derived type. For example, the following code helps explain the data types of objects used later in the program:

```
typedef float *floatp, (*float_func_p)();
```

The type `floatp` is now ‘‘pointer to a `float` value’’ type, and the type `float_func_p` is ‘‘pointer to a function returning `float`’’.

A type definition can be used anywhere the full type name is normally used (you can, of course, use the normal type name). Type definitions share the same name space as variables, and defined types are fully compatible with their equivalent types. Types defined as qualified types inherit their type qualifications.

Type definitions can also be built from other type definitions. For example:

```
typedef char byte;
typedef byte ten_bytes[10];
```

Type definition can apply to variables or functions. It is illegal to mix type definitions with other type specifiers. For example:

```
typedef int *int_p;
typedef unsigned int *uint_p;
unsigned int_p x;           /*  Invalid   */
uint_p y;                   /*  Valid     */
```

Type definitions can also be used to declare function types. However, the type definition cannot be used in the function's definition. The function's return type can be specified using a type definition. For example:

```
typedef unsigned *uint_p;   /* uint_p has type "pointer to unsigned int"    */
uint_p xp;
typedef uint_p func(void);  /* func has type "function returning pointer to */
                            /* unsigned int                                 */
func f;
func b;
  func f(void)              /* Invalid -- this declaration specifies a      */
                            /* function returning a function type, which    */
  {                         /* is not allowed                               */
    return xp;
  }

uint_p b(void)              /* Legal -- this function returns a value of
  {                         /* type uint_p.                                 */
   return xp;
  }
```

The following example shows that a function definition cannot be inherited from a `typedef` name:

```
typedef int func(int x);
func f;
func f         /*  Valid definition of f with type func                 */
{
  return 3;
}              /* Invalid, because the function's type is not inherited  */
```

Changing the previous example to a valid form results in the following:

```
typedef int func(int x);
func f;
int f(int x)   /*  Valid definition of f with type func          */
{
  return 3;
}              /* Legal, because the function's type is specified */
```

You can include prototype information, including parameter names, in the `typedef` name. You can also redefine `typedef` names in inner scopes, following the scope rules explained in Section 2.3.

4
Declarations

Declarations are used to introduce the identifiers used in a program and to specify their important attributes, such as type, storage class, and identifier name. A declaration that also causes storage to be reserved for an object or that includes the body of a function, is called a *definition*.

Section 4.1 covers general declaration syntax rules, Section 4.2 discusses initialization, and Section 4.3 describes external declarations.

The following kinds of identifiers can be declared. See the associated section for information on specific declaration and initialization syntax. Functions are discussed in Chapter 5.

- Simple objects (Section 4.4)
- Enumeration constants (Section 4.5)
- Pointers (Section 4.6)
- Arrays (Section 4.7)
- Structure and union members (Section 4.8)
- Tags (Section 4.10)

Note

Preprocessor macros created with the `#define` directive are not declarations. Chapter 8 has information on creating macros with preprocessor directives.

4.1 Declaration Syntax Rules

The general syntax of a declaration is as follows:

declaration:

declaration-specifiers init-declarator-list$_{opt}$*;*

declaration-specifiers:

storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$

init-declarator-list:

init-declarator
init_declarator-list , init-declarator

init-declarator:

declarator
declarator = initializer

Note the following items about the general syntax of a declaration:

- The *storage-class-specifier*, *type-qualifier*, and *type-specifier* can be listed in any order. All are optional, but, except for function declarations, at least one such specifier or qualifier must be present. Placing the storage-class-specifier anywhere but at the beginning of the declaration is an obsolete style.
- Storage-class keywords are `auto`, `static`, `extern`, and `register`.
- Type qualifiers are `const` and `volatile`.
- The *declarator* is the name of the object or function being declared. A declarator can be as simple as a single identifier, or can be a complex construction declaring an array, structure, pointer, union, or function (such as `*x`, `tree()`, and `treebar[10]`).

  A *full declarator* is a declarator that is not part of another declarator. The end of a full declarator is a sequence point. If the nested sequence of declarators in a full declarator contains a variable-length array type, the type specified by the full declarator is said to be *variably modified*.
- *Initializers* are optional and provide the initial value of an object. Initializers can be a single value or a brace-enclosed list of values, depending on the type of object being declared.
- A declaration determines the beginning of an identifier's scope.

- An identifier's linkage is determined by the declaration's placement and its specified storage class.

Consider the following example:

```
volatile static int data = 10;
```

This declaration shows a qualified type (a data type with a type qualifier—in this case, `int` qualified by `volatile`), a storage class (`static`), a declarator (`data`), and an initializer (`10`). This declaration is also a definition, because storage is reserved for the data object `data`.

The previous example is simple to interpret, but complex declarations are more difficult. See your platform-specific HP C documentation for more information about interpreting C declarations.

The following semantic rules apply to declarations:

- Empty declarations are illegal; declarations must contain at least one declarator, or specify a structure tag, union tag, or the members of an enumeration.
- Each declarator declares one identifier. There is no limit to the number of declarators in a declaration.
- At most, one storage-class specifier can be used in each object declaration. If none is provided, the `auto` storage class is assigned to objects declared inside a function definition, and the `extern` class is assigned to objects declared outside of a function definition.
- The only allowable (and optional) storage class for declaration of a function with block scope is `extern`.
- If no type-specifier is present, the default is `signed int`.
- A declarator is usable only over a certain range of the program, determined by the declarator's scope. The duration of its storage allocation is dependent on its storage class. See Section 2.3 for more information on scope and Section 2.10 for more information on storage classes.
- The usefulness of an identifier can be limited by its visibility, which can be hidden in some parts of the program. See Section 2.4 for more information on visibility.
- All declarations in the same scope that refer to the same object or function must have compatible types.

- If an object has no linkage, there can be no more than one declaration of the object with the same scope and in the same name space. Objects without linkage must have their type completed by the end of the declaration, or by the final initializer (if it has one). Section 2.8 describes linkage.

Storage Allocation

Storage is allocated to a data object in the following circumstances:

- If the object has no linkage, storage is allocated upon declaration of the object. If a block scope object with `auto` or `register` storage class is declared, storage is deallocated at the end of the block.
- If the object has internal linkage, storage is allocated upon the first definition of the object.
- If the object has external linkage, storage is allocated upon initialization of the object, which must occur only once for each object. If an object has only a tentative definition (see Section 2.9), the compiler acts as though there were a file scope definition of the object with an initializer of zero. Section 2.8 describes linkage in detail.

Note

The compiler does not necessarily allocate distinct variables to memory locations according to the order of declaration in the source code. Furthermore, the order of allocation can change as a result of seemingly unrelated changes to the source code, command-line options, or from one version of the compiler to the next - it is essentially unpredictable. The only way to control the placement of variables relative to each other is to make them members of the same `struct` type or, on OpenVMS Alpha systems, by using the `noreorder` attribute on a named `#pragma extern_model strict_refdef`.

4.2 Initialization

Initializers provide an initial value for objects, and follow this syntax:

initializer:

assignment-expr
{ initializer-list }
{ initializer-list, }

initializer-list:

designation-opt initializer
initializer-list, designation-opt initializer

designation:

designator-list =

designator-list:

designator
designator-list designator

designator:

[*constant-expr*]
. *identifier*

Initialization of objects of each type is discussed in the following sections, but a few universal constraints apply to all initializations in C:

- The number of initializers cannot exceed the number of objects to be initialized. Initializers can number less than the number of objects to be initialized, in which case the remaining objects are initialized to zero.
- Constant expressions must be used in an initializer for an object that has static storage duration, or in an initializer list for an object that has an aggregate or union type.
- If an identifier's declaration has block scope, and the identifier has external or internal linkage, the declaration of the identifier cannot include an initializer.
- If an object that has static storage duration is not explicitly initialized, it is initialized implicitly as if every member with an arithmetic type were assigned 0, and every member with a pointer type were assigned a null pointer constant. If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.
- The initializer for a scalar object must be a single expression, optionally enclosed in braces. The initial value of the object is that of the expression. The same type constraints and conversions apply as for simple assignment.
- If an aggregate object contains members that are aggregates or unions, or if the first member of a union is an aggregate or union, the initialization rules apply recursively to the aggregate members or contained unions. If an initializer list is used for an aggregate member or contained union, the initializers in that list initialize the members of the aggregate member or contained union. Otherwise, only enough initializers from the list are used

to account for the object; any remaining members in the list are left to initialize the next member of the aggregate object. For example:

```
struct t1 {
 int i;
 double d;
};

union t2 {
 int i;
 double d;
};

struct t3 {
 struct t1 s;
 union t2 u;
};

struct t3 st[] = { /* complete initializer */
 1, 2, 0, 4, 0, 0, 7, 0, 0
};
```

Given the previous declarations, the variable `st` is an array of 3 structures. Its initial contents are:

```
            s       u
          ------    -
st[0]:   1, 2.0,   0
st[1]:   4, 0.0,   0
st[2]:   7, 0.0,   0
```

This variable can also be defined in the following ways—all four initializers are equivalent:

```
struct t3 st[] = { /* partial initializer */
 1, 2, 0, 4, 0, 0, 7
};

struct t3 st[] = { /* nested and complete initializers */
 {1, 2, 0},
 {4, 0, 0},
 {7, 0, 0}
};

struct t3 st[] = { /* nested and partial initializers */
 {1, 2},
 {4},
 {7}
};
```

For initialization of arrays, structures, and unions, see Sections 4.7.1, 4.8.4, and 4.8.5.

- For a description of initializers with designations for arrays and structures, see Section 4.9.
- Variant structures and unions are initialized just like normal structures and unions. See Section 4.8.4 and Section 4.8.5 for more information.

C has historically allowed initializers to be optionally surrounded by extra braces (to improve formatting clarity, for instance). These initializers are parsed differently depending on the type of parser used. HP C uses the parsing technique specified by the ANSI standard, known as the top-down parse. Programs depending on a bottom-up parse of partially braced initializers can yield unexpected results. The compiler generates a warning message when it encounters unnecessary braces in common C compatibility mode or when the error-checking compiler option is specified on the command line.

4.3 External Declarations

An object declaration outside of a function is called an *external declaration*. Contrast this with an *internal declaration*, which is a declaration made inside a function or block; the declaration is internal to that function or block, and is visible only to that function or block. The compiler recognizes an internally declared identifier from the point of the declaration to the end of the block.

If an object's declaration has file scope and an initializer, the declaration is also an *external definition* for the object. A C program consists of a sequence of external definitions of objects and functions.

Any definition reserves storage for the entity being declared. For example:

```
float fvalue = 15.0;                /* external definition  */
main ()
{
  int ivalue = 15;                  /* internal definition  */
}
```

External data declarations and external function definitions take the same form as any data or function declaration (see Chapter 5 for standard function declaration syntax), and must follow these rules:

- The storage class of an object externally declared can be left unspecified, or it can be declared as `extern` or `static` (see Section 2.10). If it is unspecified, the default is the `extern` storage class, and linkage for the declared object is external. The type specifier may also be omitted, in which case the default type is `int`. Note that the *storage-class-specifier*, *type-qualifier*, and *type-specifier* cannot all be omitted from a declaration.

- If an object with external linkage is declared or used in an expression, there must be only one external definition for the identifier somewhere in the program. If the same object is declared more than once externally, the declarations must agree in type and linkage. (See Section 2.8.)
- If one or more of the declarations incompletely specify the object's type, and there exists one declaration of the object with completed type, all the declarations are taken to be in agreement with the completed type.
- The scope of external declarations persist to the end of the file in which they are declared, while internal declarations persist only to the end of the block in which they were declared. Data objects to be used within only one block should be declared in that block. The syntax for external definitions is the same as for all definitions. Function definitions can only occur at the external level.
- Externally declared `auto` and `register` objects are not permitted. Internally declared `auto` and `register` objects are not automatically initialized and, if not explicitly initialized, have the irrelevant value previously stored at their address. All `static` objects are automatically initialized to 0, if not explicitly initialized.

Note

An external function can be called without previously declaring it in C, but this construction is not recommended because of the loss of type checking and subsequent susceptibility to bugs. If such a function call is made, the compiler will treat the function as if an external declaration of type `int` appeared in the block containing the call. For example:

```
void function1()
{
int a,b;
x (a,b);
}
```

Here, the compiler will behave as if the declaration `extern int x();` appeared within the `function1` definition block.

The first declaration of an identifier in a compilation unit must specify, explicitly or by the omission of the `static` keyword, whether the identifier is internal or external. For each object, there can be only one definition. Multiple declarations of the same object may be made, as long as there are no conflicting or duplicate definitions for the same object.

An external object may be defined with either an explicit initialization or a *tentative definition*. A declaration of an object with file scope, without an initializer, and with a storage-class specifier other than `static` is a tentative definition. The compiler will treat a tentative definition as the object's only definition unless a complete definition for the object is found. As with all declarations, storage is not actually allocated until the object is defined.

If a compilation unit contains more than one tentative definition for an object, and no external definition for the object, the compiler treats the definition as if there were a file scope declaration of the object with an initializer of zero, with composite type as of the end of the compilation unit. See Section 2.7 for a definition of composite type.

If the declaration of an object is a tentative definition and has internal linkage, the declared type must not be an incomplete type. See Section 2.9 for examples of tentative definitions.

4.4 Declaring Simple Objects

Simple objects are objects with one of the basic data types. Therefore, a simple object can have an integral or floating-point type. Like all objects, simple objects are named storage locations whose values can change throughout the execution of the program. All simple objects used in a program must be declared.

A simple object declaration can be composed of the following items:

- Optional data-type specifier keywords
- Optional type-qualifier keywords (`const` or `volatile`). For example:

```
const int *p;   /*  const qualifies the integer p points to   */
int *const p;   /*  const qualifies the pointer p             */
```

- An optional storage-class keyword. If the storage-class keyword is omitted, there is a default storage class that depends on the location of the declaration in the program. The positions of the storage-class keywords and the data-type keywords are interchangeable, but placing the storage-class keyword anywhere but at the beginning of the declaration is an obsolete construction.
- Declarators, which list the names of the declared objects.
- Initializers giving the initial value of a simple object. An initializer for a simple object consists of an equal sign (=) followed by a single expression.

4.4.1 Initializing Simple Objects

An initializer for a simple object consists of an equal sign (=) followed by a single constant expression. For example:

```
int x = 10;
float y = ((12 - 2) + 25);
```

Here, the declaration both declares and defines the object x as an integer value initially equal to 10, and declares and defines the floating-point value y with an initial value of 35.

Without an initializer, the initial value of an auto object is undefined. A static object without explicit initialization is automatically initialized to 0. (If the object is a static array or structure, all members are initialized to 0.)

A block scope identifier with external or internal linkage (that is, declared using the extern or static keywords) cannot include an initializer in the declaration, because it is initialized elsewhere.

4.4.2 Declaring Integer Objects

Integer objects can be declared with the int, long, short, signed, and unsigned keywords. char can also be used, but only for small values. The following statements are examples of integer declarations:

```
int x;        /*  Declares an integer variable x    */
int y = 10;   /*  Declares an integer variable y    */
              /*  and sets y's initial value to 10  */
```

Some of the keywords can be used together to explicitly state the allowed value range. For example:

```
unsigned long int a;
signed long;           /*  Synonymous with "signed long int"   */
unsigned int;
```

Consider the range of values an integer object must be capable of representing when selecting the integral data type for the object. See Chapter 3 for more information on the size and range of integral data types.

4.4.3 Declaring Character Variables

Character objects are declared with the char keyword. The following example shows a character declaration with the initialization of a character object:

```
char ch = 'a'; /* Declares an object ch with an initial value 'a' */
```

In C, character string literals are stored in arrays of type char. See Section 4.7 for more information on arrays.

4.4.4 Declaring Floating-Point Variables

When declaring floating-point objects, determine the amount of precision needed for the stored object. Single-precision or double-precision objects can be used. For single precision, use the `float` keyword. For double precision, use the `double` or `long double` keywords. For example:

```
float x = 7.5;
double y = 3.141596;
```

See your platform-specific HP C documentation for specific information on the range and precision of floating-point types.

4.5 Declaring Enumerations

An enumerated type is a user-defined integer type. An enumerated type defines *enumeration constants*, which are integral constant expressions with values that can be represented as integers. An enumerated type declaration follows this syntax:

enum-specifier:

enum *identifier*$_{opt}$ *{ enumerator-list}*
enum *identifier*$_{opt}$ *{ enumerator-list , }*
enum *identifier*

enumerator-list:

enumerator
enumerator-list, enumerator

enumerator:

enumeration-constant
enumeration-constant = constant_expression

In HP C, objects of type `enum` are compatible with objects of type `signed int`.

The following example shows the declaration of an enumeration type and an enumeration tag:

```
enum shades
   {
      off, verydim, dim, prettybright, bright
   }  light;
```

This declaration defines the variable `light` to be of an enumerated type `shades`. `light` can assume any of the enumerated values.

The tag shades is the enumeration tag of the new type. off through bright are the enumeration constants with values 0 through 4. These enumeration constants are constant values and can be used wherever integer constants are valid.

Once a tag is declared, it can be used as a reference to that enumerated type, as in the following declaration, where the variable light1 is an object of the enumerated data type shades:

```
enum  shades  light1;
```

An incomplete type declaration of an enumerated type is illegal; for example:

```
enum e;
```

An enum tag can have the same spelling as other identifiers in the same program in other name spaces. However, enum constant names share the same name space as variables and functions, so they must have unique names to avoid ambiguity.

Internally, each enumeration constant is associated with an integer constant; the compiler gives the first enumeration constant the value 0 by default, and the remaining enumeration constants are incremented by 1 for each succeeding value. Any enumeration constant can be set to a specific integer constant value. The enumeration constants following such a construct (unless they are also set to specific values) then receive values that are one greater than the previous value. Consider the following example:

```
enum spectrum
   {
      red, yellow = 4, green, blue, indigo, violet
   }  color2 = yellow;
```

This declaration gives red, yellow, green, blue, . . . , the values 0, 4, 5, 6, . . . Assigning duplicate values to enumeration constants is permitted.

The value of color2 is an integer (4), not a string such as "red" or "yellow".

4.6 Declaring Pointers

Pointers are variables that contain the memory addresses of objects or functions. Pointer variables are declared as a pointer type by using the asterisk punctuator and the data type of the object pointed to, as shown in the following syntax:

pointer:

* *type-qualifier-list*$_{opt}$
* *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:

type-qualifier
type-qualifier-list type-qualifier

By default, HP C pointers are 32 bits long on OpenVMS systems and 64 bits long on Tru64 UNIX systems. Although their defaults are different, both OpenVMS Alpha and Tru64 UNIX systems support 32-bit (short) and 64-bit (long) pointers. HP C provides qualifiers/switches and `#pragma` preprocessor directives to control pointer size.

The type-qualifier is either `const`, `volatile`, `__unaligned` *(Alpha)*, `__restrict`, or any combination thereof.

An object of pointer type is declared as in the following example:

```
char *px;
```

In this example, identifier `px` is declared as a pointer to an object of type `char`. No type-qualifier is used in this example. The expression `*px` yields the `char` that `px` points to.

The following declarations show the difference between a variable pointer to a constant, a constant pointer to a variable, and a constant pointer to a constant object.

```
const int *ptr_to_constant;     /*  pointer variable pointing
                                    to a const object         */
int *const constant_ptr;        /*  constant pointer to a
                                    non-const object          */
const int *const constant_ptr;  /*  Const pointer to a
                                    const object              */
```

The contents of an object pointed to by `ptr_to_constant` cannot be modified through that pointer, but `ptr_to_constant` itself can be changed to point to another `const`-qualified object. Similarly, the contents of the integer pointed to by `constant_ptr` can be modified, but `constant_ptr` itself will always point to the same location.

The declaration of the constant pointer `constant_ptr` can be clarified by including a definition for the type pointer to `int`. The following example declares `constant_ptr` as an object with type *const-qualified pointer* to `int`. The pointer's value (an address) is constant:

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

The __unaligned data-type qualifier can be used in pointer definitions on Alpha systems. to indicate to the compiler that the data pointed to is not properly aligned on a correct address. (To be properly aligned, the address of an object must be a multiple of the size of the type. For example, 2-byte objects must be aligned on even addresses.) *(Alpha)*

When data is accessed through a pointer declared __unaligned, the compiler generates the additional code necessary to copy or store the data without causing alignment errors. It is best to avoid use of misaligned data altogether, but in some cases the usage may be justified by the need to access packed structures, or by other considerations. *(Alpha)*

The __restrict data-type qualifier is used to designate a pointer as pointing to a distinct object, thus allowing compiler optimizations to be made (see Section 3.7.4).

Unless an extern or static pointer variable is explicitly initialized, it is initialized to a null pointer. A null pointer is a pointer value of 0. The contents of an uninitialized auto pointer are undefined.

4.6.1 Declaring void Pointers

A void pointer is a pointer without a specified data type to describe the object to which it points. In effect, it is a generic pointer. (Before the ANSI C standard, char * was used to define generic pointers; this practice is now discouraged by the ANSI standard because it is less portable.)

A pointer to any type can be assigned to a void pointer without a cast, and vice versa. See Section 6.4.6 for more information on the cast operation. The following statements show how a void pointer can be assigned to other typed pointers, without explicit casts:

```
float *float_pointer;
void  *void_pointer;
   .
   .
   .
float_pointer = void_pointer;
                                        /*    or,      */
void_pointer  = float_pointer;
```

A void pointer is often used in function calls, function arguments, or function prototypes when a parameter or return value is a pointer of an unknown type. Consider the following example, where a void pointer is used as a generic return value:

```
void *memcpy (void *s1, const void *s2, size_t n);
{
   void  *generic_pointer;
   .
   .
   .
/* The function return value can be a pointer to many types. */

generic_pointer = func_returning_pointer( arg1, arg2, arg3 );
   .
   .
   .
/*  size_t is a defined type                                  */
}
```

See Section 5.3 for further information about using `void` in function declarations.

4.6.2 Initializing Pointers

The pointer object can be initialized with a single expression. For example:

```
int i = 10;
int *p = &i;  /*  p is a pointer to int, initialized */
              /*  as holding the address of i        */
```

Without an initializer, the values of `static` and `extern` pointers are automatically initialized to null pointers (pointers to memory location 0).

The following declaration defines `p` with type pointer to `char`, and initializes `p` to point to an object of type array of `char` with length 4, whose elements are initialized by a character string literal. (The null character is the fourth member of the array.) If an attempt is made to use `p` to modify the contents of the array, the behavior is undefined.

```
char *p =  "abc";
```

4.7 Declaring Arrays

Arrays are declared with the bracket punctuators [ ], as shown in the following syntax:

storage-class-specifier$_{opt}$ *type-specifier declarator* [* or *constant-expression-list*$_{opt}$]

The following example shows a declaration of a 10-element array of integers, a variable called `table_one`:

```
int table_one[10];
```

The *type-specifier* shows the data type of the elements. The elements of an array can be of any scalar or aggregate data type. The identifier `table_one` specifies the name of the array. The constant expression `10` gives the number of elements in a single dimension. Arrays in C are zero-based; that is, the first element of the array is identified with a 0 subscript, such as the one shown in the following example:

```
int x[5];
x[0] = 25; /* The first array element is assigned the value 25 */
```

The expression between the brackets in the declaration must be either the (*) punctuator or an integral constant expression with a value greater than zero.

If * is specified between the brackets, then the array type is a variable-length array type of unspecified size, which can be used only in declarations with function prototype scope.

If the size expression is an integer constant expression and the element type has a known constant size, the array type is not a variable-length array type. Otherwise, it is a variable-length array type. The size of each instance of a variable-length array type does not change during its lifetime. For more information on variable-length arrays, see Section 4.7.3.

Omitting the * or the constant expression creates an incomplete array declaration, which is useful in the following cases:

- If the array is declared external and its storage is allocated by a definition in another place, you can omit the constant expression for convenience when the array name is declared, as in the following example:

  ```
  extern int array1[];
  int first_function(void)
  {
     .
     .
     .
  }
  ```

  In a separate compilation unit:

  ```
  int array1[10];
  int second_function(void)
  {
     .
     .
     .
  }
  ```

The array size specifier may only be omitted from the first pair of brackets in a multidimensional array declaration. This is because an array's elements must have complete types, even if the array itself has an incomplete type.

- If the declaration of the array includes initializers (see sections 4.7.1 and 4.9), you can omit the size of the array, as in the following example:

```
char  array_one[] = "Shemps";
char  array_two[] = { 'S', 'h', 'e', 'm', 'p', 's', '\0' };
```

The two definitions initialize variables with identical elements. These arrays have seven elements: six characters and the null character (\0), which terminates all character strings. The size of the array is determined from the number of characters in the initializing character-string constant or initialization list. Initializing an incomplete array completes the array type. An array is completed at the end of its initializer list.

- If you use the array as a function parameter, the array must be defined in the calling function. However, the declaration of the parameter in the called function can omit the constant expression within the brackets. The address of the first element of the array is passed. Subscripted references in the called function can modify elements of the array. The following example shows how to use an array in this manner:

```
main()
{
                          /* Initialize array         */
   static char arg_str[] = "Thomas";
   int sum;
   sum = adder(arg_str); /* Pass address of first array element  */
      .
      .
      .
}

/*  adder adds ASCII values of letters in array       */

int adder( char param_string[])
{
    int i, sum = 0;       /* Incrementer and sum      */
                          /* Loop until NULL char     */
    for (i = 0; param_string[i] != '\0'; i++)
       sum += param_string[i];
    return sum;
}
```

After the function adder is called, parameter param_string receives the address of the first character of argument arg_str, which can then be accessed in adder. The declaration of param_string serves only to give the type of the parameter, not to reserve storage for it.

Array members can also be pointers. The following example declares an array of floating-point numbers and an array of pointers to floating-point numbers:

```
float fa[11], *afp[17];
```

When a function parameter is declared as an array, the compiler treats the declaration as a pointer to the first element of the array. For example, if x is a parameter and is intended to represent an array of integers, it can be declared as any one of the following declarations:

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

Note that the specified size of the array does not matter in the case of a function parameter, since the pointer always points to only the first element of the array.

C supports arrays declared as an array of arrays. These are sometimes called multidimensional arrays. Consider the following example, where variable table_one is a two-dimensional array containing 20 integers:

```
int table_one[10][2];
```

Arrays are stored in row-major order, which means the element table_one[0][0] (in the previous example) immediately precedes table_one[0][1], which in turn immediately precedes table_one[1][0].

4.7.1 Initializing Arrays

Arrays are initialized with a brace-enclosed list of constant expressions. A list of initializers for an incomplete array declaration completes the array's type and completely defines the array size. Therefore, when initializing an array of unknown size, the number of initializers in the initializer list determines the size of the array. For example, the following declaration initializes an array of three elements:

```
int x[] = { 1, 2, 3 };
```

If the array being initialized has a storage class of static, the initializers must be constant expressions.

Initializers for an array of a given size are assigned to array members on a one-to-one basis. If there are too few initializers for all members, the remaining members are initialized to 0. Listing too many initializers for a given size array is an error. For example:

```
int x[5] = { 0, 1, 2, 3, 4, 5 };    /*  error     */
```

String literals are often assigned to a `char` or `wchar_t` array. In this case, each character of the string represents one member of a one-dimensional array, and the array is terminated with the null character. When an array is initialized by a pointer to a string literal, the string literal cannot be modified through the pointer.

When initializing an array with a string literal, use quotation marks around the initializing string. For example:

```
char string[26] = { "This is a string literal." };
                  /* The braces above are optional here */
```

The terminating null character is appended to the end of the string if the size permits, as it does in this case. Another form for initializing an array with characters is the following:

```
char string[12] = {'T', 'h', 'i', 's', ' ', 'w', 'a', 'y' };
```

The preceding example creates a one-dimensional array containing the string value `"This way"`. The characters in this array can be freely modified. Remaining uninitialized array members will be automatically initialized to zero.

If the size of the array used for a string literal is not explicitly stated, its size is determined by the number of characters in the string (including the terminating null character). If the size of the array is explicitly stated, initializing the array with a string literal longer than the array is an error.

Note

There is one special case where the null character is not automatically appended to the array. This case is when the array size is explicitly specified and the number of initializers completely fills the array size. For example:

```
char c[4] = "abcd";
```

Here, the array `c` holds only the four specified characters, `a`, `b`, `c`, and `d`. No null character terminates the array.

Using the following rules, you can omit braces when initializing the members of a multidimensional arrays:

- When initializing arrays, you can omit the outermost pair of braces.
- If the initializer list includes all of the initializers for the object being initialized, you can omit the inner braces.

Consider the following example:

```
float x[4][2] = {
 { 1, 2 }
 { 3, 4 }
 { 5, 6 }
};
```

In this example, 1 and 2 initialize the first row of the array x, and the following two lines initialize the second and third rows, respectively. The initialization ends before the fourth row is initialized, so the members of the fourth row default to 0. Here is the result:

```
x[0][0] = 1;
x[0][1] = 2;
x[1][0] = 3;
x[1][1] = 4;
x[2][0] = 5;
x[2][1] = 6;
x[3][0] = 0;
x[3][1] = 0;
```

The following declaration achieves the same result:

```
float x[4][2] = { 1, 2, 3, 4, 5, 6 };
```

Here, the compiler fills the array row by row with the available initial values. The compiler places 1 and 2 in the first row (x[0]), 3 and 4 in the second row (x[1]), and 5 and 6 in the third row (x[2]). The remaining members of the array are initialized to zero.

Notes

- See Section 4.9 for a description of initializers with designations for arrays and structures.
- A variable-length array cannot be initialized.

4.7.2 Pointers and Arrays

Data objects in an array can be referenced through pointers instead of using array subscripts. The data type of such a pointer is referred to as ‘‘pointer to array of *type*’’. The array name itself behaves like a pointer, so there are several alternative methods to accessing array elements. For example:

```
int x[5] = { 0, 1, 2, 3, 4 }; /* Array x declared with five elements       */
int *p = x;                   /* Pointer declared and initialized to point */
                              /* to the first element of the array x       */
int a, b;
a = *(x + 3);                 /* Pointer x incremented by twelve bytes     */
                              /* to reference element 3 of x               */
b = x[3];                     /* b now holds the same value as a           */
```

In the previous example, `a` receives the value 3 by using the dereferencing operator (*). `b` receives the same value by using the subscripting operator. See Chapter 6 for more information on the different unary operators.

Note that the assignment of `a` was a result of incrementing the pointer to `x`. This principle, known as *scaling*, applies to all types of pointer arithmetic. In scaling, the compiler considers the size of an array element when calculating memory addresses of array members. For example, each member of the array `x` is 4 bytes long, and adding three to the initial pointer value automatically converts that addition to 3 * (the size of the array member, which in this case is 4). Therefore, the intuitive meaning of `z = *(y + 3);` is preserved.

When passing arrays as function arguments, only a pointer to the first element of the array is passed to the called function. The conversion from array type to pointer type is implicit. Once the array name is converted to a pointer to the first element of the array, you can increment, decrement, or dereference the pointer, just like any other pointer, to manipulate data in the array. For example:

```
int func(int *x, int *y) /* The arrays are converted to pointers        */
{
   *y = *(x + 4);        /* Various elements of the arrays are accessed  */
}
```

Remember that a pointer is large enough to hold only an address; a pointer into an array holds the address of an element of that array. The array itself is large enough to hold all members of the array.

When applied to arrays, the `sizeof` operator returns the size of the entire array, not just the size of the first element in the array.

4.7.3 Variable-Length Arrays

Variable-length arrays allow array objects with `auto` storage class, and array `typedefs` declared at block scope, to have bounds that are runtime-computed expressions.

Variable-length arrays also allow the declaration and definition of functions whose parameters are arrays dimensioned by other parameters (similar to Fortran assumed-shape arrays).

The following example illustrates both uses. Note that the definition of function `sub` uses prototype syntax and that the dimension parameters precede the array parameter that uses them. In order to define a function with the dimension parameters following the array parameter that uses them, the function definition must be written using using Kernighan and Ritchie C syntax (because that syntax allows the declarations of the types of the parameters to be written in a different order from the parameters themselves). Kernighan and Ritchie function definitions should generally be avoided.

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

void sub(int, int, int[*][*]);

int main(int argc, char **argv)
{
    if (argc != 3) {
        printf("Specify two array bound arguments.\n");
        exit(EXIT_FAILURE);
    }
    {
        int dim1 = atoi(argv[1]);
        int dim2 = atoi(argv[2]);
        int a[dim1][dim2];
        int i, j, k = 0;
        for (i = 0; i &lt; dim1; i++) {
            for (j = 0; j &lt; dim2; j++) {
                a[i][j] = k++;
            }
        }
        printf("dim1 = %d, dim2 = %d.",
               sizeof(a)/sizeof(a[0]),
               sizeof(a[0])/sizeof(int));

        sub(dim1, dim2, a);
        sub(dim2, dim1, a);
    }
    exit(EXIT_SUCCESS);
}
```

```
void sub(int sub1, int sub2, int suba[sub1][sub2])
{
    int i, j, k = 0;
    printf("\nIn sub, sub1 = %d, sub2 = %d.",
                       sub1,      sub2);
    for (i = 0; i &lt; sub1; i++) {
        printf("\n");
        for (j = 0; j &lt; sub2; j++) {
            printf("%4d", suba[i][j]);
        }
    }
}
```

On OpenVMS systems, variable-length arrays can often be used in place of the non-standard `alloca` intrinsic, `__ALLOCA`.

However, an important difference between `__ALLOCA` and variable-length arrays is that the storage allocated by `__ALLOCA` is not freed until return from the function, while the storage allocated for a variable-length array is freed on exit from the block in which it is allocated. If `__ALLOCA` is called within the scope of a variable-length array declaration (including within a block nested within the block containing a variable-length array declaration), then the storage allocated by that call to `__ALLOCA` is freed at the same time that the storage for the variable-length array is freed (that is, at block exit rather than at function return). The compiler issues a warning in such cases.

4.8 Declaring Structures and Unions

A *structure* consists of a list of members whose storage is allocated in an ordered sequence. A *union* consists of a sequence of members whose storage overlaps. Structure and union declarations have the same form, as follows:

struct-or-union-specifier:

struct-or-union identifier$_{opt}$ { *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$_{opt}$
type-qualifier specifier-qualifier-list $_{opt}$

struct-declarator-list:

struct-declarator
struct-declarator-list , struct-declarator

struct-declarator:

declarator
declarator$_{opt}$ *: constant-expression*

Neither a structure nor union member can have a function type or an incomplete type. Structures and unions cannot contain instances of themselves as members, but they can have pointers to instances of themselves as members. The declaration of a structure with no members is accepted; its size is zero.

Each structure or union definition creates a unique structure or union type within the compilation unit. The `struct` or `union` keywords can be followed by a tag, which gives a name to the structure or union type in much the same way that an `enum` tag gives a name to an enumerated type. The tag can then be used with the `struct` or `union` keywords to declare variables of that type without repeating a long definition.

The tag is followed by braces { } that enclose a list of member declarations. Each declaration in the list gives the data type and name of one or more members. The names of structure or union members can be the same as other variables, function names, or members in other structures or unions; the compiler distinguishes them by context. In addition, the scope of the member name is the same as the scope of the structure or union in which it appears. The structure or union type is completed when the closing brace completes the list.

An identifier used for a structure or union tag must be unique among the visible tags in its scope, but the tag identifier can be the same as an identifier used for a variable or function name. Tags can also have the same spellings as member names; the compiler distinguishes them by name space and context. The scope of a tag is the same as the scope of the declaration in which it appears.

Structures and unions can contain other structures and unions. For example:

```
struct person
   {
      char first[20];
      char middle[3];
      char last[30];
      struct         /*  Nested structure here  */
      {
          int day;
          int month;
          int year;
      } birth_date;
   }  employees, managers;
```

Structure or union declarations can take one of the following forms:

- If a declaration includes only a tag and a list of member declarations, then the list of member declarations defines the tag to be a data type by which other objects can be declared. The tag is considered a shorthand notation for the structure type. For example:

```
struct person
   {
      char first[20];
      char middle[3];
      char last[30];
   };
struct person employee;  /* The tag (person) identifies employee as */
                            a structure with members shown in       */
                            the declaration of person               */
```

- When a declaration includes a tag, a list of member declarations, and a list of identifiers, the identifiers become objects of the structure type and the tag is considered a shorthand notation, or mnemonic, for the structure type. The following example shows this:

```
struct person
   {
      char first[20];
      char middle[3];
      char last[30];
   }  employees, managers;
```

- If the tag is omitted, the structure or union definition applies only to the identifiers that follow in the declaration. For example:

```
struct
   {
      char first[20];
      char middle[3];
      char last[30];
   }  employees, managers;
```

- The tag can refer to a structure or union type defined elsewhere. The definition is then applied to the variable identifiers that follow the tag name in the declaration, as in the following example:

```
struct   person   employees, managers;
```

- Another form uses only the `struct` or `union` keyword and the tag to override other identical tags in the scope, and to reserve the tag for a later definition within a new scope. A definition within a new scope overrides any previous tag definition appearing in an outer scope. This use of declaring tags is called *tentative structure tag declaration*. Using such declarations, you can eliminate ambiguity when making a forward reference to tag identifiers. The following example shows such a case:

```
struct  A {...};  /*  Definition of external struct A              */

{
   struct A;   /*  Tentative structure tag declaration.              */
               /*  First declaration of A (in external scope) is
                   hidden. This structure will be defined later     */
   struct  inner
   {
      struct A *pointer;          /* Declare a structure pointer by */
       .                          /* forward referencing.          */
       .
       .
   };

   struct A {...};  /* Tentative declaration of internal struct A is
                       defined here.                                */
               /* External struct A is unaffected by this definition*/
}
```

In the example, the pointer to the structure defined using the tag `A` points to the internal definition of `A`, not the external definition.

4.8.1 Similarities Between Structures and Unions

Structures and unions share the following characteristics:

- Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.

- The only operators valid for use with entire structures and unions are the simple assignment (=) and `sizeof` operators. In particular, structures and unions cannot appear as operands of the equality (==), inequality (!=), or cast operators. The two structures or unions in the assignment must have the same members and member types.
- A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable; that is, the entire structure or union is copied into the corresponding parameter.

Note

When passing structures as arguments, they might or might not terminate on a longword boundary. If they do not, HP C aligns the following argument on the next longword boundary.

4.8.2 Differences Between Structures and Unions

The difference between structures and unions lies in the way their members are stored and initialized, as follows:

- Within a structure, the members have addresses that increase as the declarators are read left-to-right. That is, the members of a structure all begin at different offsets from the base of the structure. The offset of a particular member corresponds to the order of its declaration; the first member is at offset 0.

  A pointer to a structure points to its first member, so no unnamed holes can reside at the beginning of a structure.

  On OpenVMS VAX systems, nonbit-field structure members are byte-aligned by default. However, the `#pragma [no]member_alignment` and `#pragma pack` preprocessor directives are provided to switch from byte preprocessor directive is provided to switch from byte alignment to natural alignment.

  On Alpha systems, nonbit-field structure members are naturally aligned; each successive nonbit-field structure member begins at the next byte boundary that matches the alignment appropriate to its type. For example, a short integer is aligned on a 2-byte boundary and a long integer is aligned on a 4-byte boundary, so there may be unnamed holes in a structure.

The length of a naturally-aligned structure on a Alpha processors must be a multiple of the greatest alignment requirement of any of its members. For example, a structure containing characters, short integers, and longwords will be a multiple of four in length to match the multiple of four bytes for the longword.

The (`#pragma [no]member_alignment`) and `#pragma pack` preprocessor directives are also supported on this platform.

See your platform-specific HP C documentation for specific structure alignment requirements and examples.

- In a union, every member begins at offset 0 from the address of the union. The size of the union in memory is the size of its largest member. The value of only one member can be stored in a union object at a time. When the storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered. The rules for alignment of union members are the same as for structure members (see your platform-specific HP C documentation).

  A pointer to a union member, converted to the proper type, points to the beginning of the union object.

- Several members of a structure can be initialized at once; only the first member of a union can be given an initializer.

4.8.3 Bit Fields

One of the advantages of structures is the ability to pack data into them bit-by-bit.

A structure member often is an object with a basic type size. However, you can also declare a structure member that is composed only of a specified number of bits. Such a member is called a *bit field*; its length, an integral nonnegative constant expression, is set off from the field name by a colon, as shown by the following syntax:

struct-declarator:

declarator: constant-expression
:constant-expression

Bit fields provide greater control over the structure's storage allocation and allow tighter packing of information in memory. By using bit fields, data can be densely packed into storage.

A bit field's type must be specified (except with unnamed bit fields), and a bit field can have the `int`, `unsigned int`, or `signed int` type. The bit field's value must be small enough to store in an object of the declared size.

In the compiler's default mode, the `enum`, `long`, `short`, and `char` types are also allowed for bit fields.

A bit field can be named or unnamed. A bit-field declaration without a declarator (for example, `:10`) indicates an unnamed bit field, which is useful for padding a structure to conform to a specified layout. If the bit field is assigned a width of 0, it indicates that no further bit fields should be placed in the alignment unit, and it cannot name a declarator. Use a colon (:) to separate the member's declarator (if any) from a constant expression that gives the field width in bits. No field can be longer than 32 bits (1 longword).

Since nonbit-field structure members are aligned on at least byte boundaries, the unnamed form can create unnamed gaps in the structure's storage. As a special case, an unnamed field of width 0 causes the next member (normally another field) to be aligned on at least a byte boundary; that is, a bit-field structure member with zero width indicates that no further bit field should be packed into an alignment unit.

The following restrictions apply to the use of bit fields:

- You cannot declare arrays of bit fields.
- The ampersand operator (&) cannot be applied to fields, so there cannot be pointers to bit fields.

Sequences of bit fields are packed as tightly as possible. In C, bit fields are assigned from right to left; that is, from low-order to high-order bit.

To create bit fields, specify an identifier, a colon, and the identifier's width (in bits) as a structure member. In the following example, three bit fields are created in the structure declaration:

```
struct {
 unsigned int a : 1;  /*  Named bit field (a)    */
 unsigned int   : 0;  /*  Unnamed bit field = 0  */
 unsigned int   : 1;  /*  Unnamed bit field      */
}  class;
```

The first and third bit fields are one bit wide, the second is zero bits wide, which forces the next member to be aligned on a natural or byte boundary.

Bit fields (including zero-length bit fields) not immediately declared after other bit fields have the alignment requirement imposed by their type, but never a lesser alignment requirement than that of `int`. In a declaration of a bit field that immediately follows another bit field, the bits are packed into adjacent

space in the same alignment unit, if sufficient space remains; otherwise, padding is inserted and the second bit field is put into the next alignment unit.

See your HP C documentation for platform-specific information on bit-field alignment within a structure.

4.8.4 Initializing Structures

All structures can be initialized with a brace-enclosed list of component initializers. Structures with automatic storage class can also be initialized by an expression of compatible type.

Initializers are assigned to components on a one-to-one basis. If there are fewer initializers than members for a structure, the remaining members are initialized to 0. Listing too many initializers for the number of components in a structure is an error. All unnamed structure or union members are ignored during initialization.

Separate initializing values with commas and delimit them with braces { }. The following example initializes two structures, each with two members:

```
struct
   {
      int i;
      float c;
   }  a = { 1, 3.0e10 },  b = { 2, 1.5e5 };
```

The compiler assigns structure initializers in increasing member order. Note that there is no way to initialize a member in the middle of a structure without also initializing the previous members. Example 4–1 shows the initialization rules applied to an array of structures.

Example 4–1 The Rules for Initializing Structures

```
#include <stdio.h>
```

(continued on next page)

Example 4–1 (Cont.) The Rules for Initializing Structures

```
main()
{
   int m, n;
   static struct
      {
         char ch;
         int i;
         float c;
      }  ar[2][3] =
❶        {
❷           {
❸              { 'a', 1, 3e10 },
               { 'b', 2, 4e10 },
               { 'c', 3, 5e10 },
            }
         };

   printf("row/col\t ch\t i\t      c\n");
   printf("-----------------------------------\n");
   for (n = 0; n < 2; n++)
      for (m = 0; m < 3; m++)
         {
            printf("[%d][%d]:", n, m);
            printf("\t %c \t %d \t %e \n",
                   ar[n][m].ch, ar[n][m].i, ar[n][m].c);
         }
}
```

Key to Example 4–1:

❶ Delimit an array row initialization with braces.

❷ Delimit a structure initialization with braces.

❸ Delimit an array initialization with braces.

Example 4–1 writes the following output to the standard output:

```
row/col  ch      i            c
-------------------------------------
[0][0]:  a       1       3.000000e+10
[0][1]:  b       2       4.000000e+10
[0][2]:  c       3       5.000000e+10
[1][0]:          0       0.000000e+00
[1][1]:          0       0.000000e+00
[1][2]:          0       0.000000e+00
```

Note

See Section 4.9 for a description of initializers with designations for arrays and structures.

4.8.5 Initializing Unions

Unions are initialized with a brace-enclosed initializer that initializes only the first member of the union. For example:

```
static union
  {
     char ch;
     int i;
     float c;
  } letter = {'A'};
```

Unions with the `auto` storage class may also be initialized with an expression of the same type as the union. For example:

```
main ()
{
union1 {
    int i;
    char ch;
    float c;
  } number1 = { 2 };

auto union2
  {
    int i;
    char ch;
    float c;
  } number2 = number1;
}
```

4.9 Initializers with Designations

In conformance with the C99 standard, HP C supports the use of *designations* in the initialization of arrays and structures. (Note that designations are not supported in the common C, VAX C, and Strict ANSI89 modes of the compiler.)

4.9.1 Current Object

C99 initializers introduce the concept of a *current object* and a *designation*.

The current object is the next thing to be initialized during the initialization of an array or structure.

A designation provides a way to set the current object. When no designations are present, subobjects of the current object are initialized in order according to the type of the object: array elements in increasing subscript order, and structure members in declaration order.

So for an array, the first current object is `a[0]` when initialization begins; as each initializer is used, the current object is bumped to the next initializer, in increasing subscript order.

Similarly, for a structure, the current object is the first declaration within the structure when initialization begins; as each initializer is used, the current object is bumped to the next initializer, in declaration order.

4.9.2 Designations

The C99 Standard allows brace-enclosed initializer lists to contain *designations*, which specify a new current object. The syntax for a designation is:

```
designation:
        designator-list =

designator-list:
        designator
        designator-list designator

designator:
        [ constant-expression ]
        . identifier
```

A designator within a designation causes the following initializer to begin initialization of the object described by the designator. Initialization then continues forward, in order, beginning with the next object after that described by the designator.

For an array, a designator looks like this:

[*integral-constant-expression*]

If the array is of unknown size, any nonnegative value is valid.

For a structure, a designator looks like this:

.identifier

Where *identifier* is a member of the structure.

4.9.3 Examples

The old way of initializing arrays and structures is still supported. However, the use of designators can simplify coding of initializer lists and better accommodate future changes you might want to make to arrays and structures in your application.

1. Using designators, array elements can be initialized to nonzero values without depending on their order:

```
int a[5] = { 0, 0, 0, 5 };  // Old way
int a[5] = { [3]=5 };       // New way
```

   The designator [3] initializes a[3] to 5.

2. Structure members can be initialized to nonzero values without depending on their order. For example:

```
 typedef struct {
      char flag1;
      char flag2;
      char flag3;
       int data1;
       int data2;
       int data3;
       } Sx;
Sx = { 0, 0, 0, 0, 6 };   // Old way
Sx = { .data2 = 6 };      // New way
```

   Designator `.data2` initializes structure member .data2 to 6.

3. Another example of using designators in an array:

```
int a[10] = { 1, [5] = 20, 10 };
```

In this example, the array elements are initialized as follows:

```
a[0]=1
a[1] through a[4] = 0
a[5] = 20
a[6] = 10
a[7] through a[9] = 0
```

4. Future changes to structures can be accommodated without changing their initializer lists:

```
 typedef struct {
      char flag1;
      char flag2;
      char flag3;
       int data1;
       int data2;
       int data3;
       } Sx;

Sx = { 1, 0, 1, 65, 32, 18 };   // Old way

Sx = { .flag1=1, 0, 1, .data1=65, 32, 18 }; // New way
```

   Use of designators `.flag1` and `.data1` allows for future insertion of additional flags in front of .flag1 or between flag3 and data1.

   Designators do not have to be in order. For example, the following two initializer lists are equivalent:

```
Sx = { .data1=65, 32, 18, .flag1=1, 0, 1 };

Sx = { .flag1=1, 0, 1, .data1=65, 32, 18 };
```

5. Space can be "allocated" from both ends of an array by using a single designator:

```
int a[MAX] =
{
    1, 3, 5, 7, 9, [MAX - 5] = 8, 6, 4, 2, 0
};
```

   In this example, if `MAX` is greater than 10, there will be some zero-valued elements in the middle; if it is less than 10, some of the values provided by the first five initializers will be overridden by the second five.

6. Designators can be nested:

```
struct { int a[3], b } w[] =
{ [0].a = {1}, [1].a[0] = 2 };
```

This initialization is equivalent to the following:

```
w[0].a[0]=1;
w[1].a[0]=2;
```

7. Another example of nesting designators:

```
struct {
     int a;
     struct {
          int b
          int c[10]
     }x;
}y = {.x = {1, .c = {[5] = 6, 7 }}}
```

This initialization is equivalent to the following:

```
y.x.b = 1;
y.x.c[5] = 6;
y.x.c[6] = 7;
```

4.10 Declaring Tags

The following syntax declares the identifier *tag* as a structure, union, or enumeration tag. If this tag declaration is visible, a subsequent reference to the tag substitutes for the declared structure, union, or enumerated type. Subsequent references of the tag in the same scope (visible declarations) must omit the bracketed list. The syntax of a tag is:

struct *tag { declarator-list }*

union *tag { declarator-list }*

enum *tag { enumerator-list }*

If the tag is declared without the complete structure or union declaration, it refers to an incomplete type. Incomplete enumerated types are illegal. An incomplete type is valid only to specify an object where the type is not required; for example, during type definitions and pointer declarations. To complete the type, another declaration of the tag in the same scope (but not within an enclosed block), defines the content.

The following construction uses the tag `test` to define a self-referencing structure.

```
struct test {
 float height;
 struct test *x, *y, *z;
};
```

Once this declaration is given, the following declaration declares s to be an object of type struct test and sp to be a pointer to an object of type struct test:

```
struct test s, *sp;
```

Note

The keyword typedef can also be used in an alternative construction to do the same thing:

```
typedef struct test tnode;
struct test {
     float height;
     tnode *x, *y, *z;
};
tnode s, *sp;
```

4.11 Declaring Type Definitions

In a declaration whose storage-class specifier is typedef, each declarator defines a typedef name that specifies an alias for the stated type. A typedef declaration does not introduce a new type, but only introduces a synonym for the stated type. For example:

```
typedef int integral_type;
integral_type x;
```

In the previous example, integral_type is defined as a synonym for int, and so the following declaration of x declares x to be of type int. Type definitions are useful in cases where a long type name (such as some forms of structures or unions) benefits from abbreviation, and in cases where the interpretation of the type can be made easier through a type definition.

A typedef name shares the same name space as other identifiers in ordinary declarators. If an object is redeclared in an inner scope, or is declared as a member of a structure or union in the same or inner scope, the type specifiers cannot be omitted from the inner declaration. For example:

```
typedef signed int t;
typedef int plain;
struct tag {
   unsigned t:4;
   const t:5;
   plain r:5;
};
```

It is evident that such constructions are obscure. The previous example declares a typedef name t with type signed int, a typedef name plain with type int, and a structure with three bit-field members, one named t, another unnamed member, and a third member named r. The first two bit-field declarations differ in that unsigned is a type specifier, which forces t to be the name of a structure member by the rule previously given. The second bit-field declaration includes const, a type qualifier, which only qualifies the still-visible typedef name t.

The following example shows additional uses of the typedef keyword:

```
typedef int miles, klicksp(void);
typedef struct { double re, im; } complex;
   .
   .
   .
miles distance;
extern klicksp *metricp;
complex x;
complex z, *zp;
```

All of the code shown in the previous example is valid. The type of distance is int, the type of metricp is a pointer to a function with no parameters returning int, and the type of x and z is the specified structure. zp is a pointer to the structure.

It is important to note that any type qualifiers used with a typedef name become part of the type definition. If the typedef name is later qualified with the same type qualifier, an illegal construction results. For example:

```
typedef const int x;
const x y;            /*  Illegal -- duplicate qualifier used  */
```

5
Functions

A C program is a collection of user-defined and system-defined functions. Functions provide a convenient way to break large computing tasks into smaller ones, which helps in designing modular programs that are easier to understand and maintain. A *function* contains zero or more statements to be executed when it is called, can be passed zero or more arguments, and can return a value.

This chapter discusses the following information about C functions:

- Function calls (Section 5.1)
- Function types (Section 5.2)
- Function definitions (Section 5.3)
- Function declarations (Section 5.4)
- Function prototypes (Section 5.5)
- Parameters and arguments (Section 5.6)

5.1 Function Calls

A *function call* is a primary expression, usually a function identifier followed by parentheses, that is used to invoke a function. The parentheses contain a (possibly empty) comma-separated list of expressions that are the arguments to the function. The following is an example of a call to the function `power`, assuming this function is appropriately defined:

```
main()
{
   .
   .
   .
y = power(x,n);                    /* function call */
}
```

See Section 6.3.2 for more information on function calls.

5.2 Function Types

A function has the derived type ‘‘function returning *type*’’. The *type* can be any data type except array types or function types, although pointers to arrays and functions can be returned. If the function returns no value, its type is ‘‘function returning `void`’’, sometimes called a *void function.* A void function in C is equivalent to a procedure in Pascal or a subroutine in FORTRAN. A non-void function in C is equivalent to a function in these other languages.

Functions can be introduced into a program in one of two ways:

- A *function definition* can create a function designator, define its parameters and their type, define the type of its return value, and supply the body of the function. In the following example, `power` is a function returning `int`:

```
int power(int base, int exp)
{
 int n=1;

 if (exp < 0)
 {
  printf ("Error: Cannot handle negative exponent\n");
  return -1;
 }

 for ( ; exp; exp--)
     n = base * n;

 return n;
}
```

  See Section 5.3 for more information on function definitions.

- A *function declaration* announces the properties of a function defined elsewhere. In the following example, the function `main` declares and calls the function `power`; the definition of the function, where the code is defined, exists elsewhere:

```
main()
{
int power(int base, int exp);       /* function declaration  */
int x, n, y;
   .
   .
   .
y = power(x,n);                     /* function call         */
}
```

This style of function declaration, in which the parameters are declared in a parameter type list, is called a *function prototype*. Function prototypes require the compiler to check function arguments for consistency with their parameters, and to convert arguments to the declared types of the parameters.

See Sections 5.4 and 5.5 for more information on function declarations and prototypes.

5.3 Function Definitions

A function definition includes the code for the function. Function definitions can appear in any order, and in one source file or several, although a function cannot be split between files. Function definitions cannot be nested.

A function definition has the following syntax:

***function-definition*:**

declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

declaration-specifiers

The *declaration-specifiers* (*storage-class-specifier*, *type-qualifier*, and *type-specifier*) can be listed in any order. All are optional.

By default, the *storage-class-specifier* is `extern`. The `static` specifier is also allowed. See Section 2.10 for more information on storage-class specifiers.

ANSI allows the *type-qualifier* to be `const` or `volatile`, but either qualifier applied to a function return type is meaningless, because functions can only return rvalues and the type qualifiers apply only to lvalues.

The *type-specifier* is the data type of the value returned by the function. If no return type is specified, the function is declared to return a value of type `int`. A function can return a value of any type except ''array of *type*'' or ''function returning *type*''. Pointers to arrays and functions can be returned. The value returned, if any, is specified by an expression in a `return` statement. Executing a `return` statement terminates function execution and returns control to the calling function. For functions that return a value, any expression with a type compatible with the function's return type can follow `return` using the following format:

return *expression*;

If necessary, the expression is converted to the return type of the function. Note that the value returned by a function is not an lvalue. A function call, therefore, cannot constitute the left side of an assignment operator.

The following example defines a function returning a character:

```
char letter(char param1)
{
   .
   .
   .
   return param1;
}
```

The calling function can ignore the returned value. If no expression is specified after `return`, or if a function terminates by encountering the right brace, then the return value of the function is undefined. No value is returned in the case of a `void` function.

If a function does not return a value, or if the function is always called from within a context that does not require a value, a return type of `void` should be specified:

```
void message()
{
   printf("This function has no return value.");
   return;
}
```

Specifying a return type of `void` in a function definition or declaration generates an error under the following conditions:

- If the function attempts to return a value, an error occurs at the offending `return` statement.
- If the `void` function is called in a context that requires a value, an error occurs at the function call site.

declarator

The *declarator* specifies the name of the function being declared. A declarator can be as simple as a single identifier, such as `f1` in the following example:

```
int f1(char p2)
```

In the following example, `f1` is a ‘‘function returning `int`’’. A declarator can also be a more complex construct, as in the following example:

```
int (*(*fpapfi(int x))[5])(float)
```

In this example, `fpapfi` is a ‘‘function (taking an `int` argument) returning a pointer to an array of five pointers to functions (taking a `float` argument) returning `int`’’. See Chapter 4 for information on specific declarator syntax.

The declarator (function) need not have been previously declared. If the function was previously declared, the parameter types and return type in the function definition must be identical to the previous function declaration.

The declarator can include a list of the function's parameters. In HP C, up to 253 parameters can be specified in a comma-separated list enclosed in parentheses. Each parameter has the `auto` storage class by default, although `register` is also allowed. There is no semicolon after the right parenthesis of the parameter list.

There are two methods of specifying function parameters:

- The *new* or *prototype* style, which includes a parameter type list. For example:

```
int f1(char a, int b)
{
function body
}
```

- The *old* style, which includes an identifier list; the parameter types are defined in a separate *declaration-list* within the function definition, before the left brace that begins the function body. For example:

```
int f1(a, b)
char a;
int b;
{
function body
}
```

  Any undeclared parameters are assumed to be of type `int`.

A function definition with no parameters is defined with an empty parameter list. An empty parameter list is specified in either of two ways:

- Using the keyword `void` if the prototype style is used. For example:

```
char msg(void)
{
return 'a';
}
```

- Using empty parentheses if the old style is used. For example:

```
char msg()
{
return 'a';
}
```

A function defined using the prototype style establishes a prototype for that function. The prototype must agree with any preceding or following declarations of the same function.

A function defined using the old style does not establish a prototype, but if a prototype exists because of a previous declaration for that function, the parameter declarations in the definition must exactly match those in the prototype after the default argument promotions are applied to the parameters in the definition.

Avoid mixing old style and prototype style declarations and definition for a given function. It is allowed but not recommended.

See Section 5.6 for more information on function parameters and arguments. See Section 5.5 for more information on function prototypes.

compound-statement

The *compound-statement* is the group of declarations and statements surrounded by braces in a function or loop body. This compound statement is also called the *function body*. It begins with a left brace ({) and ends with a right brace (}), with any valid C declarations and statements in between. One or more `return` statements can be included, but they are not required.

5.4 Function Declarations

A function can be called without declaring it if the function's return value is `int` (although this practice is not recommended due to the loss of type-checking capability; all functions should be declared). If the return value is anything else, and if the function definition is located after the calling function in the source code, the function must be declared before calling it. For example:

```
char lower(int c);                    /* Function declaration */

caller()                              /* Calling function     */
{
int c;
char c_out;
      .
      .
      .
c_out = lower(c);                     /* Function call        */

}
```

```
char lower(int c_up)                    /* Function definition  */
{
   .
   .
   .
}
```

If the function definition for `lower` was located before the function `caller` in the source code, `lower` would not have to be declared again before calling it. In that case, the function definition would serve as its own declaration and would be in scope for any function calls from within all subsequently defined functions in the same source file.

Note that both the function definition and function declaration for `lower` are in the prototype style. Although C supports the old style of function declaration in which the parameter types are not specified in the function declarator, it is good programming practice to use prototype declarations for all user-defined functions in your program, and to place the prototypes before the first use of the function. Also note that it is valid for the parameter identifier in the function declaration to be different from the parameter identifier in the function definition.

In a function declaration, the `void` keyword should be used to specify an empty argument list. For example:

```
char function_name(void);
```

As with function definitions, the `void` keyword can also be used in function declarations to specify the return value type for functions that do not return a value. For example:

```
main()
{
   void function_name( );
      .
      .
      .
}
void function_name( )
{ }
```

5.5 Function Prototypes

A function prototype is a function declaration that specifies the data types of its arguments in the parameter list. The compiler uses the information in a function prototype to ensure that the corresponding function definition and all corresponding function declarations and calls within the scope of the prototype contain the correct number of arguments or parameters, and that each argument or parameter is of the correct data type.

Prototypes are syntactically distinguished from the old style of function declaration. The two styles can be mixed for any single function, but this is not recommended. The following is a comparison of the old and the prototype styles of declaration:

Old style:

- Functions can be declared implicitly by their appearance in a call.
- Arguments to functions undergo the default conversions before the call.
- The number and type of arguments are not checked.

Note

The HP C compiler will warn about old-style function declarations only in strict ANSI standard mode, or when the check compiler option is specified.

Prototype style:

- Functions are declared explicitly with a prototype before they are called. Multiple declarations must be compatible; parameter types must agree exactly.
- Arguments to functions are converted to the declared types of the parameters.
- The number and type of arguments are checked against the prototype and must agree with or be convertible to the declared types. Empty parameter lists are designated using the `void` keyword.
- Ellipses are used in the parameter list of a prototype to indicate that a variable number of parameters are expected.

5.5.1 Prototype Syntax

A function prototype has the following syntax:

***function-prototype-declaration*:**

declaration-specifiers$_{opt}$ *declarator*;

The *declarator* includes a parameter type list, which specifies the types of, and can declare identifiers for, the parameters of the function.

A parameter type list can consist of a single parameter of type `void` to specify that the function has no parameters.

A parameter type list can contain a member that is a variable-length array, specified by the `[*]` notation.

In its simplest form, a function prototype declaration might have the following format:

storage_class$_{opt}$ *return_type*$_{opt}$ *function_name* (*type*$_1$ *parameter*$_1$, ..., *type*$_n$ *parameter*$_n$);

Consider the following function definition:

```
char  function_name( int lower, int *upper, char (*func)(), double y )
{ }
```

The corresponding prototype declaration for this function is:

```
char  function_name( int lower, int *upper, char (*func)(), double y );
```

A prototype is identical to the header of its corresponding function definition specified in the prototype style, with the addition of a terminating semicolon (;) or comma (,), as appropriate (depending on whether the prototype is declared alone or in a multiple declaration).

Function prototypes need not use the same parameter identifiers as in the corresponding function definition because identifiers in a prototype have scope only within the identifier list. Moreover, the identifiers themselves need not be specified in the prototype declaration; only the types are required.

For example, the following prototype declarations are equivalent:

```
char  function_name( int lower, int *upper, char (*func)(), double y );
char  function_name( int a, int *b, char (*c)(), double d );
char  function_name( int, int *, char (*)(), double );
```

Though not required, identifiers should be included in prototypes to improve program clarity and increase the type-checking capability of the compiler.

Variable-length argument lists are specified in function prototypes with ellipses. At least one parameter must precede the ellipses. For example:

```
char  function_name( int lower, ... );
```

Data-type specifications cannot be omitted from a function prototype.

The C99 standard permits the keyword `static` to be used within the outermost array bound of a formal parameter in a prototype function declaration. The effect is to assert to the compiler that at each call to the function, the corresponding actual argument will provide access to at least as many array elements as specified in the declared array bound. Consider the following two function definitions:

```
void foo(int a[1000]){ ... }
void bar(int b[static 1000]) { ...}
```

The declaration of `foo` is absolutely equivalent to one that declares `a` to be `int *`. When compiling the body of `foo`, the compiler has no information about how many array elements might exist. The declaration of `bar` differs in that the compiler can assume that at least 1000 array elements exist and may be safely accessed. The intent is to provide a hint to the optimizer about what can be safely pre-fetched.

5.5.2 Scope and Conversions

Prototypes must be placed appropriately in each compilation unit of a program. The position of the prototype determines its scope. A function prototype, like any function declaration, is considered within the scope of a corresponding function call only if the prototype is specified within the same block as the function call, any enclosing block, or at the outermost level of the source file. The compiler checks all function definitions, declarations, and calls from the position of the prototype to the end of its scope. If you misplace the prototype so that a function definition, declaration, or call occurs outside the scope of the prototype, any calls to that function behave as if there were no prototype.

The syntax of the function prototype is designed so that you can extract the function header of each of your function definitions, add a semicolon (`;`), place the prototypes in a header, and include that header at the top of each compilation unit in your program. In this way, function prototypes are declared to be external, extending the scope of the prototype throughout the entire compilation unit. To use prototype checking for C library function calls, place the `#include` preprocessor directives for the `.h` files appropriate for the library functions used in the program.

It is an error if the number of arguments in a function definition, declaration, or call does not match the prototype.

If the data type of an argument in a function call does not match the corresponding type in the function prototype, the compiler tries to perform conversions. If the mismatched argument is assignment-compatible with the prototype parameter, the compiler converts the argument to the data type specified in the prototype, according to the argument conversion rules (see Section 5.6.1).

If the mismatched argument is not assignment-compatible with the prototype parameter, an error message is issued.

5.6 Parameters and Arguments

C functions exchange information by means of parameters and arguments. The term *parameter* refers to any declaration within the parentheses following the function name in a function declaration or definition; the term *argument* refers to any expression within the parentheses of a function call.

The following rules apply to parameters and arguments of C functions:

- Except for functions with variable-length argument lists, the number of arguments in a function call must be the same as the number of parameters in the function definition. This number can be zero.
- The maximum number of arguments (and corresponding parameters) is 253 for a single function.
- Arguments are separated by commas. However, the comma is not an operator in this context, and the arguments can be evaluated by the compiler in any order. There is, however, a sequence point before the actual call.
- Arguments are passed by value; that is, when a function is called, the parameter receives a copy of the argument's value, not its address. This rule applies to all scalar values, structures, and unions passed as arguments.
- Modifying a parameter does not modify the corresponding argument passed by the function call. However, because arguments can be addresses or pointers, a function can use addresses to modify the values of variables defined in the calling function.
- In the old style, parameters that are not explicitly declared are assigned a default type of `int`.
- The scope of function parameters is the function itself. Therefore, parameters of the same name in different functions are unrelated.

5.6.1 Argument Conversions

In a function call, the types of the evaluated arguments must match the types of their corresponding parameters. If they do not match, the following conversions are performed in a manner that depends on whether a prototype is in scope for the function:

- Arguments to functions specified with prototypes are converted to the parameter types specified in the prototype, except that arguments corresponding to an ellipsis (...) are converted as if no prototype were in scope. (In this case, the rules in the following bullet apply.) For example:

  ```
  void f(char, short, float, ...);

  char c1, c2;
  short s1,s2;
  float f1,f2;

  f(c1, s1, f1, c2, s2, f2);
  ```

  The arguments `c1`, `s1`, and `f1` are passed with their respective types, while the arguments `c2`, `s2`, and `f2` are converted to `int`, `int`, and `double`, respectively.

- Arguments to functions that have no prototype in scope are not converted to the types of the parameters. Instead, the expressions in the argument list are converted according to the following rules:

  - Any arguments of type `float` are converted to `double`.
  - Any arguments of types `char`, `unsigned char`, `short`, or `unsigned short` are converted to `int`.
  - When compiling in common C compatibility mode, HP C converts any arguments of types `unsigned char` or `unsigned short` to `unsigned int`.

No other default conversions are performed on arguments. If a particular argument must be converted to match the type of the corresponding parameter, use the cast operator. For more information about the cast operator, see Section 6.4.6.

5.6.2 Function and Array Identifiers as Arguments

Function and array identifiers can be specified as arguments to a function. Function identifiers are specified without parentheses, and array identifiers are specified without brackets. When so specified, the function or array identifier is evaluated as the address of that function or array. Also, the function must be declared or defined, even if its return value is an integer. Example 5–1 shows how and when to declare functions passed as arguments, and how to pass them.

Example 5–1 Declaring Functions Passed as Arguments

```
❶ int x()  { return 25; }            /* Function definition and   */
  int z[10];                         /* array defined before use  */

❷ fn(int f1(), int (*f2)(), int a1[]))   /* Function definition      */
  {
      f1();                          /* Call to function f1       */
        .
        .
        .
  }

  void caller(void)
  {
❸    int y();                        /* Function declaration      */
        .
        .
        .
❹    fn(x, y, z);                    /* Function call: functions  */
                                     /* x and y, and array z      */
                                     /* passed as addresses       */
        .
        .
        .
  }
  int y(void) { return 30; }         /* Function definition       */
```

Key to Example 5–1:

❶ Without being declared in a separate declaration, function x can be passed in an argument list because its definition, located before the function `caller`, serves as its declaration.

❷ Parameters that represent functions can be declared either as functions or as pointers to functions. Parameters that represent arrays can be declared either as arrays or as pointers to the element type of the array. For example:

```
fn(int f1(), int f2(), int a1[])      /* f1, f2 declared as     */
{...}                                 /* functions; a1 declared */
                                      /* as array of int.       */


fn(int (*f1)(), int (*f2)(), int *a1) /* f1, f2 declared as     */
{...}                                 /* pointers to functions; */
                                      /* a1 declared as pointer */
                                      /* to int.                */
```

When such parameters are declared as functions or arrays, the compiler automatically converts the corresponding arguments to pointers.

❸ Because its function definition is located after the function `caller`, function `y` must be declared before passing it in an argument list.

❹ When passing functions as arguments, do not include parentheses. Similarly, when specifying arrays, do not include subscripts.

5.6.3 Passing Arguments to the main Function

The function called at program startup is named `main`. The `main` function can be defined with no parameters or with two parameters (for passing command-line arguments to a program when it begins executing). The two parameters are referred to here as *argc* and *argv*, though any names can be used because they are local to the function in which they are declared. A `main` function has the following syntax:

```
int main(void) { . . . }

int main(int argc, char *argv[ ]) { . . . })
```

argc

The number of arguments in the command line that invoked the program. The value of *argc* is nonnegative.

argv

Pointer to an array of character strings that contain the arguments, one per string. The value `argv[`*argc*`]` is a null pointer.

If the value of *argc* is greater than zero, the array members *argv*[0] through *argv*[*argc* − 1] inclusive contain pointers to strings, which are given implementation-defined values by the host environment before program startup. The intent is to supply the program with information determined before program startup from elsewhere in the host environment. If the host environment cannot supply strings with letters in both uppercase and lowercase, the host environment ensures that the strings are received in lowercase.

If the value of *argc* is greater than zero, the string pointed to by *argv*[0] represents the program name; *argv*[0][0] is the null character if the program name is not available from the host environment. If the value of *argc* is greater than one, the strings pointed to by *argv*[1] through *argv*[*argc* − 1] represent the program parameters.

The parameters *argc* and *argv*, and the strings pointed to by the *argv* array, can be modified by the program and keep their last-stored values between program startup and program termination.

In the main function definition, parameters are optional. However, only the parameters that are defined can be accessed.

See your platform-specific HP C documentation for more information on the passing and return of arguments to the `main` function.

6
Expressions and Operators

An expression is any sequence of C operators and operands that produces a value or generates a side effect. The simplest expressions are constants and variable names, which yield values directly. Other expressions combine operators and subexpressions to produce values. An expression has a type as well as a value.

Except where noted in this chapter, the order of evaluation of subexpressions, and the order in which side effects take place, is unspecified. Code that depends on such order might produce unexpected results.

The operands of expressions must have compatible type. In some instances, the compiler makes conversions to force the data types of the operands to be compatible.

The following sections discuss these topics:

- Primary expressions and operators (Section 6.1)
- An overview of the C operators (Section 6.2)
- Postfix expressions (Section 6.3)
- Unary expressions and operators (Section 6.4)
- Binary expressions and operators (Section 6.5)
- The conditional expression and operator (Section 6.6)
- Assignment expressions and operators (Section 6.7)
- The comma expression and operator (Section 6.8)
- Constant expressions (Section 6.9)
- Compound literal expressions (Section 6.10)
- Data-type conversions (Section 6.11)

6.1 Primary Expressions

Simple expressions are called primary expressions; they denote values. Primary expressions include previously declared identifiers, constants, string literals, and parenthesized expressions.

Primary expressions have the following syntax:

***primary-expression*:**

identifier
constant
string-literal
expression

The following sections describe the primary expressions.

6.1.1 Identifiers

An identifier is a primary expression provided it is declared as designating an object or a function.

An identifier that designates an object is an lvalue if its type is arithmetic, structure, union, or pointer. The name of an array evaluates to the address of the first element of the array; an array name is an lvalue but not a modifiable lvalue.

An identifier that designates a function is called a *function designator*. A function designator evaluates to the address of the function.

6.1.2 Constants

A constant is a primary expression. Its type depends on its form (integer, character, floating, or enumeration); see Section 1.9. A constant is never an lvalue.

6.1.3 String Literals

A string literal is a primary expression. Its type depends on its form (character or `wchar_t`); see Section 1.9. A string literal is an lvalue.

6.1.4 Parenthesized Expressions

An expression within parentheses has the same type and value as the expression without parentheses would have. Any expression can be delimited by parentheses to change the precedence of its operators.

6.2 Overview of the C Operators

Variables and constants can be used in conjunction with C operators to create more complex expressions. Table 6–1 presents the set of C operators.

Table 6–1 C Operators

Operator	Example	Description
()	`f()`	Function call
[]	`a[10]`	Array reference
->	`s->a`	Structure and union member selection
.	`s.a`	Structure and union member selection
+ [unary]	`+a`	Value of a
- [unary]	`-a`	Negative of a
* [unary]	`*a`	Reference to object at address a
& [unary]	`&a`	Address of a
~	`~a`	One's complement of a
++ [prefix]	`++a`	The value of a after increment
++ [postfix]	`a++`	The value of a before increment
-- [prefix]	`--a`	The value of a after decrement
-- [postfix]	`a--`	The value of a before decrement
`sizeof`	`sizeof (t1)`	Size in bytes of object with type t1
`sizeof`	`sizeof e`	Size in bytes of object having the type of expression e
`__typeof__`	`__typeof__ (t1)`	Type of type t1
`__typeof__`	`__typeof__ (e)`	Type of expression e
`_Pragma`	`_Pragma` (*string-literal*)	destringize *string-literal*
+ [binary]	`a + b`	a plus b
- [binary]	`a - b`	a minus b
* [binary]	`a * b`	a times b
/	`a / b`	a divided by b
%	`a % b`	Remainder of a/b
>>	`a >> b`	a, right-shifted b bits
<<	`a << b`	a, left-shifted b bits

(continued on next page)

Table 6–1 (Cont.) C Operators

Operator	Example	Description
<	a < b	1 if a < b; 0 otherwise
>	a > b	1 if a > b; 0 otherwise
<=	a <= b	1 if a <= b; 0 otherwise
>=	a >= b	1 if a >= b; 0 otherwise
==	a == b	1 if a equal to b; 0 otherwise
!=	a != b	1 if a not equal to b; 0 otherwise
& [binary]	a & b	Bitwise AND of a and b
|	a | b	Bitwise OR of a and b
^	a ^ b	Bitwise XOR (exclusive OR) of a and b
&&	a && b	Logical AND of a and b (yields 0 or 1)
||	a || b	Logical OR of a and b (yields 0 or 1)
!	!a	Logical NOT of a (yields 0 or 1)
?:	a ? e1 : e2	Expression e1 if a is nonzero; Expression e2 if a is zero
=	a = b	a, after b is assigned to it
+=	a += b	a plus b (assigned to a)
-=	a -= b	a minus b (assigned to a)
*=	a *= b	a times b (assigned to a)
/=	a /= b	a divided by b (assigned to a)
%=	a %= b	Remainder of a/b (assigned to a)
>>=	a >>= b	a, right-shifted b bits (assigned to a)
<<=	a <<= b	a, left-shifted b bits (assigned to a)
&=	a &= b	a AND b (assigned to a)
|=	a |= b	a OR b (assigned to a)
^=	a ^= b	a XOR b (assigned to a)
,	e1,e2	e2 (e1 evaluated first)

The C operators fall into the following categories:

- Postfix operators, which follow a single operand.
- Unary prefix operators, which precede a single operand.
- Binary operators, which take two operands and perform a variety of arithmetic and logical operations.
- The conditional operator (a ternary operator), which takes three operands and evaluates either the second or third expression, depending on the evaluation of the first expression.
- Assignment operators, which assign a value to a variable.
- The comma operator, which guarantees left-to-right evaluation of comma-separated expressions.

Operator *precedence* determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

```
x = 7 + 3 * 2;        /* x is assigned 13, not 20   */
```

The previous statement is equivalent to the following:

```
x = 7 + ( 3 * 2 );
```

Using parenthesis in an expression alters the default precedence. For example:

```
x = (7 + 3) * 2;  /*  (7 + 3) is evaluated first    */
```

In an unparenthesized expression, operators of higher precedence are evaluated before those of lower precedence. Consider the following expression:

```
A+B*C
```

The identifiers `B` and `C` are multiplied first because the multiplication operator (`*`) has higher precedence than the addition operator (`+`).

Table 6–2 shows the precedence the compiler uses to evaluate the C operators. Operators with the highest precedence appear at the top of the table; those with the lowest appear at the bottom. Operators of equal precedence appear in the same row.

Table 6–2 Precedence of C Operators

Category	Operator	Associativity
Postfix	`( )  [ ]  ->  .  ++  --`	Left to right

Unary	`+  -  !  ~  ++  --`
	`(type)`
	`*  &  sizeof`
		Right to left

Multiplicative	`*  /  %`	Left to right
Additive	`+  -`	Left to right
Shift	`<<  >>`	Left to right
Relational	`<  <=  >  >=`	Left to right
Equality	`==  !=`	Left to right
Bitwise AND	`&`	Left to right
Bitwise XOR	`^`	Left to right

(continued on next page)

Table 6–2 (Cont.) Precedence of C Operators

Category	Operator	Associativity
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Associativity relates to precedence, and resolves any ambiguity over the grouping of operators with the same precedence. In the following statement, the rules of C specify that `a * b` is evaluated first:

```
y = a * b / c;
```

In a more complicated example, associativity rules specify that `b ? c : d` is evaluated first in the following example:

```
a ? b ? c : d : e;
```

The associativity of the conditional operator is right-to-left on the line. The assignment operator also associates right-to-left; for example:

```
int x = 0 , y = 5, z = 3;
x = y = z;                  /*  x has the value 3, not 5    */
```

Other operators associate left-to-right; for example, the binary addition, subtraction, multiplication, and division operators all have left-to-right associativity.

Associativity applies to each row of operators in Table 6–2 and is right-to-left for some rows and left-to-right for others. The kind of associativity determines the order in which operators from the same row are evaluated in an unparenthesized expression. Consider the following expression:

```
A*B%C
```

This expression is evaluated as follows because the multiplicative operators (*, /, %) are evaluated from left to right:

```
(A*B)%C
```

Parentheses can always be used to control precedence and associativity within an expression.

6.3 Postfix Operators

Postfix expressions include array references, function calls, structure or union references, and postfix increment and decrement expressions. The operators in postfix expressions have left-to-right associativity.

Postfix expressions have the following syntax:

***postfix-expression*:**

array-reference
function-call
structure-and-union-member-reference
postfix-increment-expression
postfix-decrement-expression

6.3.1 Array References

The bracket operator [ ] is used to refer to an element of an array. Array references have the following syntax:

***array-reference*:**

postfix-expression [*expression*]

For example, in a one-dimensional array, you can refer to a specific element within the array as follows:

```
int sample_array[10];  /* Array declaration; array has 10 elements */
sample_array[0] = 180; /* Assign value to first array element      */
```

This example assigns a value of 180 to the first element of the array, `sample_array[0]`. Note that C uses zero-origin array subscripting.

In a two-dimensional array (more properly termed an array of arrays), you can refer to a specific element within the array, as follows:

```
int sample_array[10][5];  /* Array declaration; array has 50 elements */
sample_array[9][4] = 180; /* Assign value to last array element       */
```

This example assigns a value of 180 to the element `sample_array[9][4]`.

Conceptually, multidimensional arrays are of type *arrays of arrays of arrays ....* Therefore, if an array reference is not fully qualified, it refers to the *address* of the first element in the dimension that is not specified. For example:

```
int sample_array[10][5]; /* Array declaration                    */
int *p1;                 /* Pointer declaration                  */

p1 = sample_array[7];    /* Assigns address of subarray to pointer */
```

In this example, p1 contains the address of the first element in the one-dimensional subarray sample_array[7]. Although, as in this example, a partially qualified array can be used as an rvalue, only a fully qualified array reference can be used as an lvalue. For example, C does not allow the following statement, in which the second dimension of the array is omitted:

```
int sample_array[10][5]; /* Array declaration                    */
sample_array[7] = 21;    /* Error                                */
```

A reference to an array name with no bracket can be used to pass the array's address to a function, as in the following statement:

```
funct(sample_array);
```

Bracket operators can also be used to perform general pointer arithmetic as follows:

```
p1[intexp]
```

Here, p1 is a pointer and intexp is an integer-valued expression. The result of the expression is the value pointed to by p1 incremented by the value of intexp multiplied by the size, in bytes, of the addressed object (array element). The expressions *(p1 + intexp) and p1[intexp] are defined to be equivalent; both expressions refer to the same memory location and have the same type. Array subscripting is a commutative operation: intexp[p1] is equivalent to p1[intexp]. A subscripted expression is always an lvalue.

6.3.2 Function Calls

Function calls have the following syntax:

***function-call*:**

postfix-expression (*argument-expression-list*$_{opt}$)

***argument-expression-list*$_{opt}$:**

assignment-expression
argument-expression-list$_{opt}$, *assignment-expression*

A function call is a postfix expression consisting of a *function designator* followed by parentheses. The order of evaluation of any expressions in the function parameter list is undefined, but there is a sequence point before the actual call. The parentheses can contain a list of arguments (separated by commas) or can be empty. If the function called has not been declared, it is assumed to be a function returning int.

To pass an argument that is an array or function, specify the identifier in the argument list without brackets or parentheses. The compiler passes the address of the array or function to the called routine, which means that the corresponding parameters in the called function must be declared as pointers.

In the following example, `func1` is declared as a function returning `double`; the number and type of the parameters are not specified:

```
double func1();
```

The function `func1` can then be used in a function call, as follows:

```
result = func1(c);
      or
result = func1();
```

The identifier `func1` can also be used in other contexts, without the parentheses. For example, as an argument to another function call:

```
dispatch(func1);
```

In this example, the address of the function `func1` is passed to the function `dispatch`. In general, if an identifier is declared as a ''function returning . . . '' type, it is converted to ''the address of function returning . . . '' when that identifier is passed as an argument without its parentheses; the only exception is when the function designator is the operand of the unary `&` operator, in which case this conversion is explicit.

Functions can also be called by dereferencing a pointer to a function. In the following example, `pf` is declared as a pointer to a function returning `double` and assigned the address of the function `func1`:

```
double (*pf)( );
   .
   .
   .
pf = func1;
```

The function `func1` can then be called as follows:

```
result = (*pf)();
```

Although this function call is valid, the following form of the same function call is simpler:

```
result = pf();
```

In function calls, if the expression that denotes the called function has a type that does not include a prototype, the integer promotions discussed in Section 6.11.3 are performed on each applicable argument, and arguments that have type `float` are converted to `double`. These are called the *default argument promotions*. If the number of passed arguments does not agree with the number of parameters, the behavior is undefined. If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with the types of the parameters after promotion, the behavior is undefined. If the function is defined with a type that includes a prototype, and the types of the arguments after promotion are not compatible with the types of the parameters, or if the prototype ends with an ellipsis punctuator (indicating a variable-length parameter list), the behavior is undefined.

If the expression that denotes the called function has a type that includes a prototype, the passed arguments are implicitly converted to the types of the corresponding parameters. The ellipsis punctuator in a function prototype causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments. If the function is defined with a type that is not compatible with the type pointed to by the expression that denotes the called function, the behavior is undefined.

No other conversions are implicitly performed; in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a prototype.

Recursive function calls are permitted, both directly and indirectly through any chain of other functions.

6.3.3 Structure and Union References

A member of a structure or union can be referenced either directly using the dot (.) operator, or indirectly using the arrow (->) operator.

Structure and union references (also called component selections) have the following syntax:

***structure-and-union-reference*:**

postfix-expression . *identifier*
postfix-expression -> *identifier*

The arrow operator always produces an lvalue. The dot operator produces an lvalue if the postfix expression is an lvalue.

In a direct member selection, the first operand must designate a structure or union, and the *identifier* must name a declared member of that structure or union.

In an indirect member selection, the first operand must be a pointer to a structure or union, and the identifier must name a declared member of that structure or union. The arrow operator is specified with a hyphen (-) and a greater-than symbol (>) and designates a reference to the structure or union member. The expression *E1->name* is, by definition, precisely the same as (**E1*).*name*. This also implies that *E2.name* is the same as (&*E2*)->*name*, if *E2* is an lvalue.

A named structure member must be fully qualified; that is, it must be preceded by a list of the names of any higher-level members separated by periods, arrows, or both. The value of the expression is the named member of the structure or union, and its type is the type of that member. For more information about structures and unions, see Sections 3.4.4 and 3.4.5.

With one exception, if a member of a union is accessed after a value has been stored in a different member of that union, the result is dependent on the data types of the members referenced and their alignment within the union.

The exception exists to simplify the use of unions. If a union contains several structures that share a common initial sequence, and if the union currently contains one of these structures, you can inspect the common initial part of any of them. Two structures share a common initial sequence if corresponding members have compatible types (and for bit fields, the same width) for a sequence of one or more initial members.

6.3.4 Postfix Increment and Decrement Operators

C has two unary operators for incrementing and decrementing objects of scalar type. Postfix incrementation has the following syntax:

***postfix-increment-expression*:**

postfix-expression ++

Postfix decrementation has the following syntax:

postfix-decrement-expression:

***postfix-decrement-expression*:**

postfix-expression – –

The increment operator ++ adds 1 to its operand, and the decrement operator –– subtracts 1, except when the operand is a pointer. If the operand is a pointer of type *pointer to T,* the pointer is incremented (or decremented) by `sizeof(`*T*`)`. The effect is to point to the next (or previous) element within an array of objects of type *T*.

Both ++ and −− can be used either as prefix operators (before the operand: ++n) or postfix operators (after the operand: n++). In both cases, the effect is to increment n. The expression ++n increments n before its value is used, while n++ increments n after its value is used.

Section 6.4.3 describes the prefix form of the increment and decrement operators. This section addresses the postfix form.

Consider the following expression:

lvalue++

The postfix operator ++ adds the constant 1 to the operand, modifying the operand. The value of the expression is the value of the operand incremented by 1; otherwise, the result of the expression is the old value of the operand, before it was incremented. For example:

```
int i, j;
j = 5;
j++;                    /* j = 6 (j incremented by 1) */
i = j++;                /* i = 6, j = 7               */
```

When using the increment and decrement operators, do not depend on the order of evaluation of expressions. Consider the following ambiguous expression:

```
k = x[j] + j++;
```

It is unspecified whether the value of j in x[j] is evaluated before or after j is incremented. To avoid ambiguity, increment the variable in a separate statement, as in the following example:

```
j++;
k = x[j] + j;
```

The ++ and −− operators can also be used with floating-point objects. In this case they scale the object by 1.0.

6.4 Unary Operators

Unary expressions are formed by combining a unary operator with a single operand. All unary operators are of equal precedence and have right-to-left associativity. The unary operators are:

- Unary minus (−) and unary plus (+) (see Section 6.4.1)
- Logical negation (!) (see Section 6.4.2)
- Prefix increment (++) and decrement (−−) (see Section 6.4.3)
- Address operator (&) and indirection (*) (see Section 6.4.4)

- Bitwise negation (one's complement) (~) (see Section 6.4.5)
- Cast operator (see Section 6.4.6)
- `sizeof` operator (see Section 6.4.7)

6.4.1 Unary Plus and Minus

Consider the following expression:

`-- expression`

This is the negative of the operand. The operand must have an arithmetic type, and integer promotion is applied. The additive inverse of an `unsigned` quantity is computed by subtracting the quantity from the largest value of the `unsigned` type plus one.

The unary plus operator returns the value of an expression:

+ expression

Neither the unary plus nor unary minus operators produce lvalues.

6.4.2 Logical Negation

Consider the following expression:

! expression

The result is the logical (Boolean) negation of the expression. If the value of the expression is 0, the negated result is 1; if the value of the expression is not 0, the negated result is 0. The type of the result is `int`. The expression must have a scalar type.

6.4.3 Prefix Increment and Decrement Operators

C has two unary operators for incrementing and decrementing scalar objects. The increment operator ++ adds 1 to its operand; the decrement operator -- subtracts 1. Both ++ and -- can be used either as prefix operators (before the variable: `++n`) or postfix operators (after the variable: `n++`). In both cases, the effect is to increment `n`. The expression `++n` increments `n` before its value is used, while `n++` increments `n` after its value is used.

Section 6.3.4 describes the postfix increment and decrement operators. This section describes the prefix form.

Consider the following expression:

++modifiable lvalue

After evaluating this expression, the result is the incremented rvalue, not the corresponding lvalue. For this reason, expressions that use the increment and decrement operators in this manner cannot appear by themselves on the left side of an assignment expression where an lvalue is needed.

If declared as an integer or floating-point number, the operand is increased or decreased by 1 (or 1.0). The results of the following C statements are equivalent:

```
i = i + 1;
i++;
++i;
i += 1;
```

The following example shows the difference between the postfix and prefix forms of the increment operator:

```
int i, j;
j = 5;
i = ++j;                        /*  i = 6, j = 6  */
i = j++;                        /*  i = 6, j = 7  */
```

If the operand is a pointer, the address is incremented by the size of the addressed object as determined by its data type, not by the integer value 1. For example:

```
char *cp;
int  *ip;
++cp;             /* Incremented by sizeof(char) */
++ip;             /* Incremented by sizeof(int)  */
```

Consider the following expression:

-- modifiable lvalue

The prefix operator -- is similar to the prefix operator ++ except that the value of the operand is decremented.

When using the increment and decrement operators, do not depend on the order of evaluation of expressions. Consider the following ambiguous expression:

```
k = x[j] + ++j;
```

It is unspecified whether the value of `j` in `x[j]` is evaluated before or after `j` is incremented. To avoid ambiguity, increment the variable in a separate statement, as in the following example:

```
++j;
k = x[j] + j;
```

6.4.4 Address Operator and Indirection

Consider the following expression:

&lvalue

This expression results in the address of the lvalue. The lvalue can be a function designator or any lvalue that designates an object, including an unqualified array identifier. The lvalue cannot be a `register` variable or a bit field.

Consider the following expression:

**pointer*

When an expression resolves to an address, the value stored at that address can be accessed by using the dereferencing operator (*).

If the operand of * is a function name or function pointer, then the result is a function designator. If the operand of * is a pointer to an object, then the result is an lvalue designating the object. If an invalid value (0, for example) is assigned to the pointer, then the * operation is undefined.

The dereferencing operator * always produces an lvalue. The address operator & never produces an lvalue.

6.4.5 Bitwise Negation

Consider the following expression:

~ expression

The result is the bitwise negation (one's complement) of the evaluated expression. Each 1-bit is converted into a 0-bit and vice versa. The expression must have an integer type. The compiler performs the usual arithmetic conversions (see Section 6.11.1).

6.4.6 The Cast Operator

The cast operator forces the conversion of its scalar operand to a specified scalar data type, or to `void`. The operator consists of a *type-name*, in parentheses, that precedes an expression, as follows:

(type-name) expression

The value of the expression is converted to the named data type, as if the expression were assigned to a variable of that type. The expression's type and value are not themselves changed; the value is converted to the cast type for the duration of the cast operation. The *type-name* has the following syntax:

***type-name*:**

type-specifier abstract-declarator

In simple cases, *type-specifier* is the keyword for a data type, such as `char` or `double`, and *abstract-declarator* is empty. For example:

```
(int)x;
```

The *type-specifier* can also be an `enum` specifier, or a `typedef` name. The *type-specifier* can be a structure or union only if the *abstract-declarator* is a pointer. That is, the *type-name* can be a pointer to a structure or union, but cannot be a structure or union because structures and unions are not scalar types. For example:

```
(struct abc *)x   /* allowed     */
(struct abc)x     /* not allowed */
```

The *abstract-declarator* in a cast operation is a declarator without an identifier. Abstract declarators have the following syntax:

***abstract-declarator*:**

empty
abstract-declarator
* *abstract-declarator*
abstract-declarator ( )
abstract-declarator [*constant-expression*]

The *abstract-declarator* cannot be empty in the following form:

(*abstract-declarator*)

Abstract declarators can include the brackets and parentheses that indicate arrays and functions. However, cast operations cannot force the conversion of any expression to an array, function, structure, or union. The brackets and parentheses are used in operations such as the following example, which casts the identifier `P1` to *pointer to array of* `int`*:*

```
(int (*)[10]) P1;
```

This kind of cast operation does not change the contents of `P1`; it only causes the compiler to treat the value of `P1` as a pointer to such an array. For example, casting pointers this way can change the scaling that occurs when an integer is added to a pointer:

```
int *ip;
((char *)ip) + 1;   /* Increments by 1 not by 4 */
```

Cast operators can be used in the following conversions that involve pointers:

- A pointer can be converted to an integral type. A pointer occupies the same amount of storage as objects of type `int` or `long` (or their `unsigned` equivalents). Therefore, a pointer can be converted to any of these integer types and back again without changing its value. No scaling takes place, and the representation of the value does not change.

  Converting from a pointer to a shorter integer type is similar to converting from an `unsigned long` type to a shorter integer type; that is, the high-order bits of the pointer are discarded.

  Converting from a shorter integer type to a pointer is similar to the conversion from a shorter integer type to an object of `unsigned long` type; that is, the high-order bits of the pointer are filled with copies of the sign bit. HP C, with the check option enabled, issues a warning message for cast operations of this type.

- A pointer to an object or incomplete type can be converted to a pointer to a different object or a different incomplete type. The resulting pointer might not be valid if it is improperly aligned for the type pointed to. It is guaranteed, however, that a pointer to an object of a given alignment can be converted to a pointer to an object of the same alignment or less strict alignment, and back again. The result is equal to the original pointer. (An object of character type has the least strict alignment.)

- A pointer to a function of one type can be converted to a pointer to a function of another type and back again; the result is equal to the original pointer. If a converted pointer is used to call a function that has a type not compatible with the type of the called function, the behavior is undefined.

6.4.7 The sizeof Operator

Consider the syntax of the following expressions:

sizeof *expression*

sizeof (*type-name*)

type-name cannot be an incomplete type, function type, or a bit field. The `sizeof` operator produces a compile-time integer constant value. *expression* is inspected only to deduce its type; it is not fully evaluated. For example, `sizeof(i++)` is equivalent to `sizeof(i)`.

The result of the sizeof operation is the size, in bytes, of the operand. In the first case, the result of sizeof is the size determined by the type of the expression. In the second case, the result is the size of an object of the named type. The expression should be enclosed in parentheses if it contains operators, because the precedence of sizeof is higher than that of most operators.

The syntax of *type-name* is the same as that for the cast operator. For example:

```
int  x;
x = sizeof(char *);  /* assigns the size of a character pointer to x */
```

The type of the sizeof operator's result, size_t, is an unsigned integer type. In HP C, size_t is unsigned int.

6.4.8 The __typeof__ Operator

The __typeof__ operator is another way to refer to the type of an expression. This feature is provided for compatiblity with the gcc compiler.

The syntax of this operator keyword looks like sizeof, but the construct acts semantically like a type-name defined with typedef.

__typeof__ (*expression*)

__typeof__ (*type-name*)

There are two ways of writing the argument to __typeof__: with an expression or with a type.

The following is an example with an expression. This example assumes that x is an array of ints; the type described is int:

```
__typeof__(x[0](1))
```

The following is an example with a type-name as the argument. The type described is that of pointers to int:

```
__typeof__(int *)
```

A __typeof__ construct can be used anywhere a typedef name can be used. For example, you can use it in a declaration, in a cast, or inside a sizeof or __typeof__ operator:

```
__typeof__(*x) y;     // Declares y with the type of what x points to.
__typeof__(*x) y[4];  // Declares y as an array of such values.
__typeof__(__typeof__(char *)[4]) y;  // Declares y as an array of
                                      // pointers to characters:
```

The last example (the nested __typeof__ operators) is equivalent to the following traditional C declaration:

```
char *y[4];
```

To see the meaning of the declaration using __typeof__, and why it might be a useful way to write it that way, let's rewrite it with these macros:

```
#define pointer(T)  __typeof__(T *)
#define array(T, N) __typeof__(T [N])
```

Now the declaration can be rewritten this way:

```
array (pointer (char), 4) y;
```

Thus, `array (pointer (char), 4)` is the type of arrays of 4 pointers to `char`.

6.4.9 The _Pragma Operator

The _Pragma operator *destringizes* its string literal argument, effectively allowing #pragma directives to be produced by macro expansion. When specified using this operator, the tokens of the pragma, which appear together within a single string literal in this form, are not macro expanded, even if they have _m suffix. But macro expansion can be accomplished if desired by using the stringization operator (#) to form the string (see Section 8.1.3).

The _Pragma operator has the following syntax:

_Pragma (*string-literal*)

A _Pragma operator expression is processed as follows: The string literal is destringized by deleting the L prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence \" by a double-quote, and replacing each escape sequence \\ by a single backslash.

The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

Example: A directive of the form:

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ( "listing on \"..\\listing.dir\"" )
```

The latter form is processed in the same way, whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x)  _Pragma(#x)

LISTING ( ..\listing.dir )
```

6.5 Binary Operators

The binary operators are categorized as follows:

- Multiplicative operators: multiplication (*), remainder (%), and division (/) (see Section 6.5.1)
- Additive operators: addition (+) and subtraction (–) (see Section 6.5.2)
- Shift operators: left shift (<<) and right shift (>>) (see Section 6.5.3)
- Relational operators: less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=) (see Section 6.5.4)
- Equality operators: equality (==) and inequality (!=) (see Section 6.5.5)
- Bitwise operators: AND (&), OR (|), and XOR (^) (see Section 6.5.6)
- Logical operators: AND (&&) and OR (||) (see Section 6.5.7)

The following sections describe these binary operators.

6.5.1 Multiplicative Operators

The multiplicative operators are *, /, and %. Operands must have arithmetic type. Operands are converted, if necessary, according to the usual arithmetic conversion rules (see Section 6.11.1).

The * operator performs multiplication.

The / operator performs division. When integers are divided, truncation is toward zero. If either operand is negative, the result is truncated toward zero (the largest integer of lesser magnitude than the algebraic quotient).

The % operator divides the first operand by the second and yields the remainder. Both operands must be integral. When both operands are unsigned or positive, the result is positive. If either operand is negative, the sign of the result is the same as the sign of the left operand.

The following statement is true if b is not zero:

```
(a/b)*b + a%b == a;
```

The HP C compiler, with the check option enabled, issues warnings for these undefined behaviors:

- Integer overflow occurs
- Division by zero is attempted
- Remainder by zero is attempted

6.5.2 Additive Operators

The additive operators + and − perform addition and subtraction, respectively. Operands are converted, if necessary, according to the usual arithmetic conversion rules (see Section 6.11.1).

When two `enum` constants or variables are added or subtracted, the type of the result is `int`.

When an integer is added to or subtracted from a pointer expression, the integer is scaled by the size of the object being pointed to. The result has the pointer's type. For example:

```
int arr[10];
int *p = arr;
p = p + 1;  /* Increments by sizeof(int) */
```

An array pointer can be decremented by subtracting an integral value from a pointer or address; the same conversions apply as for addition. Pointer arithmetic also applies one element beyond the end of the array. For example, the following code works because the pointer arithmetic is limited to the elements of the array and to only one element beyond:

```
int i = 0;
int x[5] = {0,1,2,3,4};
int y[5];
int *ptr = x;
while (&y[i] != (ptr + 5)) { /*  ptr + 5 marks one beyond the end of the array */
  y[i] = x[i];
  i++;
}
```

When two pointers to elements of the same array are subtracted, the result (calculated by dividing the difference between the two addresses by the length of one element) is of type `ptrdiff_t`, which in HP C is `int`, and represents the number of elements between the two addressed elements. If the two elements are not in the same array, the result of this operation is undefined.

6.5.3 Shift Operators

The shift operators << and >> shift their left operand to the left or to the right, respectively, by the number of bits specified by the right operand. Both operands must be integral. The compiler performs integral promotions on each of the operands (see Section 6.11.1.1). The type of the result is the type of the promoted left operand. Consider the following expression:

E1 << *E2*

The result is the value of expression *E1* shifted to the left by *E2* bits. Bits shifted off the end are lost. Vacated bits are filled with zeros. The effect of shifting left is to multiply the left operand by 2 for each bit shifted. In the following example, the value of `i` is 100:

```
int n = 25;
int m = 2;
int i;

i = n << m;
```

Consider the following expression:

E1 >> *E2*

The result is the value of expression *E1* shifted to the right by *E2* bits. Bits shifted off the end are lost. If *E1* is `unsigned` or if *E1* has a signed type but nonnegative value, vacated bits are filled with zeros. If *E1* has a signed type and negative value, vacated bits are filled with ones.

The result of the shift operation is undefined if the right operand is negative or if its value is greater than the number of bits in an `int`.

For a nonnegative left operand, the effect of shifting right is to divide the left operand by 2 for each bit shifted. In the following example, the value of `i` is 12:

```
int n = 100;
int m = 3;
int i;

i = n >> m;
```

6.5.4 Relational Operators

The relational operators compare two operands and produce a result of type `int`. The result is 0 if the relation is false, and 1 if it is true. The operators are: less than (<), greater than (>), less than or equal (<=), and greater than or equal (>=). Both operands must have an arithmetic type or must be pointers to compatible types. The compiler performs the necessary arithmetic conversions before the comparison (see Section 6.11.1).

When two pointers are compared, the result depends on the relative locations of the two addressed objects. Pointers to objects at lower addresses are less than pointers to objects at higher addresses. If two addresses indicate elements in the same array, the address of an element with a lower subscript is less than the address of an element with a higher subscript.

The relational operators associate from left to right. Therefore, the following statement relates `a` to `b`, and if `a` is less than `b`, the result is 1 (true). If `a` is greater than or equal to `b`, the result is 0 (false). Then, 0 or 1 is compared with `c` for the expression result. This statement does not determine ''if `b` is between `a` and `c`''.

```
if ( a < b < c )
    statement;
```

To check if `b` is between `a` and `c`, use the following code:

```
if ( a < b && b < c )
    statement;
```

6.5.5 Equality Operators

The equality operators, equal (==) and not-equal (!=), produce a result of type `int`, so that the result of the following statement is 1 if both operands have the same value, and 0 if they do not:

```
a == b
```

Operands must have one of the following type combinations:

- Both operands have an arithmetic type.
- Both operands are pointers to qualified or unqualified versions of compatible types.
- One operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of `void`.
- One operand is a pointer and the other is a null pointer constant.

Operands are converted, if necessary, according to the usual arithmetic conversion rules (see Section 6.11.1).

Two pointers or addresses are equal if they identify the same storage location.

Note

Although different symbols are used for assignment (=) and equality (==), C allows either operator in all contexts, so be careful not to confuse them. Consider the following example:

```
if ( x = 1 )
    statement_1;
else
    statement_2;
```

In this example, `statement_1` always executes, because the result of the assignment `x = 1` is equivalent to the value of `x`, which equals 1 (or true).

6.5.6 Bitwise Operators

The bitwise operators require integral operands. The usual arithmetic conversions are performed (see Section 6.11.1). The result of the expression is the bitwise AND (&), inclusive OR (|), or exclusive OR (^), of the two operands. The order of evaluation of their operands is not guaranteed.

The operands are evaluated bit by bit. The result of the & operator is 0 if one bit value is 0 and the other is 1, or if both bit values are 0. The result is 1 if both bit values are 1.

The result of the | operator is 0 if both bit values are 0. The result for each bit is 1 if either bit value is 1, or both bit values are 1.

The result of the ^ operator is 0 if both bit values are 0, or if both bit values are 1. The result for each bit is 1 if either bit value is 1 and the other is 0.

6.5.7 Logical Operators

The logical operators are AND (&&) and OR (||). These operators guarantee left-to-right evaluation. The result of the expression (of type `int`) is either 0 (false) or 1 (true). The operands need not have the same type, but both types must be scalar. If the compiler can make an evaluation by examining only the left operand, the right operand is not evaluated. Consider the following expression:

E1 && E2

The result of this expression is 1 if both operands are nonzero, or 0 if one operand is 0. If expression *E1* is 0, expression *E2* is not evaluated because the result is the same regardless of *E2*'s value.

Similarly, the following expression is 1 if either operand is nonzero, and 0 otherwise. If expression *E1* is nonzero, expression *E2* is not evaluated, because the result is the same regardless of *E2*'s value.

E1 || *E2*

6.6 Conditional Operator

The conditional operator (?:) takes three operands. It tests the result of the first operand and then evaluates one of the other two operands based on the result of the first. Consider the following example:

E1 ? *E2* : *E3*

If expression *E1* is nonzero (true), then *E2* is evaluated, and that is the value of the conditional expression. If *E1* is 0 (false), *E3* is evaluated, and that is the value of the conditional expression. Conditional expressions associate from right to left. In the following example, the conditional operator is used to get the minimum of `x` and `y`:

```
a = (x < y) ? x : y;     /* a = min(x, y)  */
```

There is a sequence point after the first expression (*E1*). The following example's result is predictable, and is not subject to unplanned side effects:

```
i++ > j ? y[i] : x[i];
```

The conditional operator does not produce an lvalue. Therefore, a statement such as `a ? x : y = 10` is not valid.

The following restrictions apply:

- The first operand must have a scalar type.
- One of the following must hold for the second and third operands:
  - Both operands have an arithmetic type (the usual arithmetic conversions are performed to bring the second and third operands to a common type). The result has that type.
  - Both operands have compatible structure or union types.
  - Both operands have a type of `void`.
  - Both operands are pointers to qualified or unqualified versions of compatible types. The result has the composite type.

- One operand is a pointer and the other is a null pointer constant. The result has the type of the pointer that is not a null pointer constant.
- One operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of `void`. The result has the type *pointer to* `void`.

6.7 Assignment Operators

There are several assignment operators. Assignments result in the value of the target variable after the assignment. They can be used as subexpressions in larger expressions. Assignment operators do not produce lvalues.

Assignment expressions have two operands: a modifiable lvalue on the left and an expression on the right. A *simple assignment* consists of the equal sign (=) between two operands:

```
E1 = E2;
```

The value of expression *E2* is assigned to *E1*. The type is the type of *E1*, and the result is the value of *E1* after completion of the operation.

A *compound assignment* consists of two operands, one on either side of the equal sign (=), in combination with another binary operator. For example:

```
E1 += E2;
```

This is equivalent to the following simple assignment (except that in the compound assignment *E1* is evaluated once, while in the simple assignment *E1* is evaluated twice):

```
E1 = E1 + E2;
```

In the following example, the following assignments are equivalent:

```
a *= b + 1;
a = a * (b + 1);
```

In another example, the following expression adds 100 to the contents of `number[1]`:

```
number[1] += 100;
```

The result of this expression is the result after the addition and has the same type as `number[1]`.

If both assignment operands are arithmetic, the right operand is converted to the type of the left before the assignment (see Section 6.11.1).

The assignment operator (=) can be used to assign values to structures and unions. In the VAX C compatibility mode of HP C, one structure can be assigned to another as long as the structures are defined to be the same size, in bytes. In ANSI mode, the structure values must also have the same type. With all compound assignment operators, all right operands and all left operands must be either pointers or evaluate to arithmetic values. If the operator is -= or +=, the left operand can be a pointer, and the right operand (which must be integral) is converted in the same manner as the right operand in the binary plus (+) and minus (-) operations.

Do not reverse the characters that comprise a compound assignment operator, as in the following example:

```
E1 =+ E2;
```

This is an obsolete form that is no longer supported, but it will pass through the compiler undetected. (It is interpreted as an assignment operator followed by the unary plus operator).

6.8 Comma Operator

When two or more expressions are separated by the comma operator, they evaluate from left to right. The result has the type and value of the rightmost expression (although side effects of the other expressions, if any, do take place). The result is not an lvalue. In the following example, the value 1 is assigned to `R`, and the value 2 is assigned to `T`:

```
R = T = 1,   T += 2,   T -= 1;
```

Side effects for each expression are completed before the next expression is evaluated.

A comma expression must be enclosed with parentheses if it appears where commas have some other meaning, as in argument and initializing lists. Consider the following expression:

```
f(a, (t=3,t+2), c)
```

This example calls the function `f` with the arguments `a`, `5`, and `c`. In addition, variable `t` is assigned the value `3`.

6.9 Constant Expressions

A constant expression is an expression that contains only constants. A constant expression can be evaluated during compilation rather than at run time, and can be used in any place that a constant can occur. In the following example, `limit+1` is a constant expression, and is evaluated at compile time:

```
#define limit 500
char x[limit+1]
```

A constant expression cannot contain assignment, increment, decrement, function-call, or comma operators, except when they are within the operand of a `sizeof` operator. Each constant expression must evaluate to a constant that is in the range of representable values for its type.

There are several contexts in which C requires an expression that must evaluate to a constant:

- The size of a bit field
- The value of an enumeration constant
- The size of an array (and the second and subsequent dimensions in all array declarations)
- The value of a `case` label
- An integral constant expression used in conditional-inclusion preprocessing directives
- The initializer list for an object with static storage duration

6.9.1 Integral Constant Expressions

An integral constant expression has an integral type and contains only operands that are integer constants, enumeration constants, character constants, `sizeof` expressions whose operand does not have variable-length array type or a parenthesized name of such a type, or floating constants that are the immediate operands of casts. Cast operands in an integral constant expression only convert arithmetic types to integral types, except as part of an operand to the `sizeof` operator.

C allows more latitude for constant expressions in initializers. Such a constant expression can evaluate to one of the following:

- An arithmetic constant expression
- A null pointer constant
- An address constant

- An address constant for an object type plus or minus an integral constant expression

6.9.2 Arithmetic Constant Expressions

An arithmetic constant expression has an arithmetic type and contains only operands that are integer constants, floating constants, enumeration constants, character constants, or `sizeof` expressions whose operand does not have variable-length array type or a parenthesized name of such a type. Cast operators in an arithmetic constant expression only convert arithmetic types to arithmetic types, except as part of an operand to the `sizeof` operator.

6.9.3 Address Constants

An address constant is a pointer to an lvalue designating an object of static storage duration (see Section 2.10), or to a function designator. Address constants must be created explicitly by using the unary `&` operator, or implicitly by using an expression of array or function type. The array subscript `[ ]` and member access operators `.` and ->, the address & and indirection * unary operators, and pointer casts can be used to create an address constant, but the value of an object cannot be accessed by use of these operators.

6.10 Compound Literal Expressions

A *compound literal*, also called a *constructor expression*, is a form of expression that constructs the value of an object, including objects of array, struct, or union type.

In the C89 Standard, passing a `struct` value to a function typically involves declaring a named object of the type, initializing its members, and passing that object to the function. With the C99 Standard, this can now be done with a single compound literal expression. (Note that compound literal expressions are not supported in the common C, VAX C, and Strict ANSI89 modes of the HP C compiler.)

A compound literal is an unnamed object specified by a syntax consisting of a parenthesized type name (the same syntax as a cast operator[1]) followed by a brace-enclosed list of initializers. The value of this unnamed object is given by the initializer list. The initializer list can use the designator syntax.

For example, to construct an array of 1000 `int`s that are all zero except for array element 9, which is to have a value of 5, you can write the following:

```
(int [1000]){[9] = 5}.
```

[1] However, this differs from a cast expression in that a cast specifies a conversion to scalar types or void only, and the result of a cast expression is not an lvalue.

A compound literal object is an lvalue. The object it designates has static storage duration if it occurs outside all function definitions. Otherwise, it has automatic storage duration associated with the nearest enclosing block.

Usage Notes

- The type name must specify an object type or an array of unknown size.
- An initializer cannot provide a value for an object not contained within the entire unnamed object specified by the compound literal.
- If the compound literal occurs outside the body of a function, the initializer list must consist of constant expressions.
- If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in Section 4.7.1, and the type of the compound literal is that of the completed array type. Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. In either case, the result is an lvalue.
- All the semantic rules and constraints for initializer lists in Sections 4.2, 4.7.1, 4.8.4, 4.8.5, and 4.9 are applicable to compound literals.
- String literals, and compound literals with const-qualified types, need not designate distinct objects. This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations.

The following examples illustrate the use of compound literals.

Examples

1.
```
int *p = (int []){2, 4};
```

This example initializes `p` to point to the first element of an array of two `int`s, the first having the value 2 and the second having the value 4. The expressions in this compound literal are required to be constant. The unnamed object has static storage duration.

2.

```
void f(void)
{
        int *p;
        /*...*/
        p = (int [2]){*p};
        /*...*/
}
```

In this example, p is assigned the address of the first element of an array of two ints, the first having the value previously pointed to by p and the second having the value zero. The expressions in this compound literal need not be constant. The unnamed object has automatic storage duration.

3.

```
drawline((struct point){.x=1, .y=1},
        (struct point){.x=3, .y=4});
```

Or, if drawline instead expected pointers to struct point:

```
drawline(&(struct point){.x=1, .y=1},
        &(struct point){.x=3, .y=4});
```

Initializers with designations can be combined with compound literals. Structure objects created using compound literals can be passed to functions without depending on member order.

4.

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

A read-only compound literal can be specified through constructions like the one in this example.

5.

```
"/tmp/testfile"
(char []){"/tmp/testfile"}
(const char []){"/tmp/testfile"}
```

The three expressions in this example have different meanings:

The first always has static storage duration and has type "array of char", but need not be modifiable.

The last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

6.
```
(const char []){"abc"} == "abc"
```

Like string literals, const-qualified compound literals can be placed into read-only memory and can even be shared. This example might yield 1 if the literal's storage is shared.

7.
```
struct int_list { int car; struct int_list *cdr; };
struct int_list endless_zeros = {0, &endless_zeros};
eval(endless_zeros);
```

Because compound literals are unnamed, a single compound literal cannot specify a circularly linked object. In this example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object endless_zeros.

8.
```
struct s { int i; };

int f (void)
{
        struct s *p = 0, *q;
        int j = 0;

        while (j < 2)
                 q = p, p = &((struct s){ j++ });
        return p == q && q->i == 1;
}
```

As shown in this example, each compound literal creates only a single * object in a given scope.

The function f() always returns the value 1.

6.11 Data-Type Conversions

C performs data-type conversions in the following four situations:

- When two or more operands of different types appear in an expression.
- When arguments of type char, short, and float are passed to a function using the old style declaration.
- When arguments that do not conform exactly to the parameters declared in a function prototype are passed to a function.

- When the data type of an operand is deliberately converted by the cast operator. See Section 6.4.6 for more information on the cast operator.

The following sections describe how operands and function arguments are converted.

6.11.1 Usual Arithmetic Conversions

The following rules—referred to as the usual arithmetic conversions—govern the conversion of all operands in arithmetic expressions. The effect is to bring operands to a common type, which is also the type of the result. The rules govern in the following order:

1. If either operand is not of arithmetic type, no conversion is performed.
2. If either operand has type `long double`, the other operand is converted to `long double`.
3. Otherwise, if either operand has type `double`, the other operand is converted to `double`.
4. Otherwise, if either operand has type `float`, the other operand is converted to `float`.
5. For complex numbers (*(Alpha, I64)*), the above conversions take place for the corresponding real type of the operand to be converted.

   For example, adding a `double _Complex` and a `float`, converts just the `float` operand to `double`, yielding a `double _Complex` result.
6. Otherwise, the integer promotions are performed on both operands, and the following rules apply:

   a. If both operands have the same type, then no further conversion is needed.

   b. Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

   c. Otherwise, If either operand has type `unsigned long long int`, the other operand is converted to `unsigned long long int`.

   d. If either operand has type `unsigned long int`, the other operand is converted to `unsigned long int`.

e. Otherwise, if one operand has type `long int` and the other has type `unsigned int`, and if a `long int` can represent all values of an `unsigned int`, the operand of type `unsigned int` is converted to `long int`. If a `long int` cannot represent all the values of an `unsigned int`, both operands are converted to `unsigned long int`.

f. Otherwise, if either operand has type `long int`, the other operand is converted to `long int`.

g. Otherwise, if either operand has type `unsigned int`, the other operand is converted to `unsigned int`.

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

The following sections elaborate on the usual arithmetic conversion rules.

6.11.1.1 Characters and Integers

A `char`, `short int`, or `int` bit field, either signed or unsigned, or an object that has enumeration type, can be used in an expression wherever an `int` or `unsigned int` is permitted. If an `int` can represent all values of the original type, the value is converted to an `int`. Otherwise, it is converted to an `unsigned int`. These conversion rules are called the *integer promotions*.

This implementation of integer promotion is called *value preserving*, as opposed to *unsigned preserving* in which `unsigned char` and `unsigned short` widen to `unsigned int`. HP C uses value-preserving promotions, as required by the ANSI C standard, unless the common C mode is specified.

To help locate arithmetic conversions that depend on unsigned preserving rules, HP C, with the check option enabled, flags any integer promotions of `unsigned char` and `unsigned short` to `int` that could be affected by the value-preserving approach for integer promotions.

All other arithmetic types are unchanged by the integer promotions.

In HP C, variables of type `char` are bytes treated as signed integers. When a longer integer is converted to a shorter integer or to `char`, it is truncated on the left; excess bits are discarded. For example:

```
int i;
char c;

i = 0xFFFFFF41;
c = i;
```

This code assigns hex 41 (`'A'`) to `c`. The compiler converts shorter signed integers to longer ones by sign extension.

6.11.1.2 Signed and Unsigned Integers

Conversions also take place between the various kinds of integers.

When a value with an integral type is converted to another integral type (such as `int` converted to `long int`) and the value can be represented by the new type, the value is unchanged.

When a signed integer is converted to an unsigned integer of equal or greater size, and the signed integer value is nonnegative, its value is unchanged. If the signed integer value is negative, then:

- If the unsigned integer type is larger, the signed integer is first promoted to the signed integer that corresponds to the unsigned integer; then the value is converted to unsigned by adding to it one greater than the largest number that can be represented in the unsigned integer type.
- If the unsigned integer type is equal or smaller than the signed integer type, then the value is converted to unsigned by adding to it one greater than the largest number that can be represented in the unsigned integer type.

When an integer value is demoted to an unsigned integer of smaller size, the result is the nonnegative remainder of the value divided by the number one greater than the largest representable unsigned value for the new integral type.

When an integer value is demoted to a signed integer of smaller size, or an unsigned integer is converted to its corresponding signed integer, the value is unchanged if it is small enough to be represented by the new type. Otherwise, the result is truncated; excess high-order bits are discarded and precision is lost.

Conversion between integral types of the same size, whether signed or unsigned, results in no machine-level representation change.

6.11.1.3 Floating and Integral

When a floating-type operand is converted to an integer, the fractional part is discarded.

When a floating-type value is to be converted at compile time to an integer or another floating type, and the result cannot be represented, the compiler reports a warning in the following instances:

- The conversion is to `unsigned int` and the result cannot be represented by the `unsigned int` type.

- The conversion is to a type other than `unsigned int`, and the result cannot be represented by the `int` type.

When a value of integral type is converted to floating type, and the value is in the range of values that can be represented, but not exactly, the result of the conversion is either the next higher or next lower value, whichever is the natural result of the conversion on the hardware. See your HP C documentation for the conversion result on your platform.

6.11.1.4 Floating Types

If an operand of type `float` appears in an expression, it is treated as a single-precision object unless the expression also involves an object of type `double` or `long double`, in which case the usual arithmetic conversion applies.

When a `float` is promoted to `double` or `long double`, or a `double` is promoted to `long double`, its value is unchanged.

The behavior is undefined when a `double` is demoted to `float`, or a `long double` to `double` or `float`, if the value being converted is outside the range of values that can be represented.

If the value being converted is inside the range of values that can be represented, but not exactly, the result is rounded to either the next higher or next lower representable `float` value.

6.11.2 Pointer Conversions

Although two types (for example, `int` and `long`) can have the same representation, they are still different types. This means that a pointer to `int` cannot be assigned to a pointer to `long` without using a cast. Nor can a pointer to a function of one type be assigned to a pointer to a function of a different type without using a cast. In addition, pointers to functions that have different parameter-type information, including the old-style absence of parameter-type information, are different types. In these instances, if a cast is not used, the compiler issues an error. Because there are alignment restrictions on some target processors, access through an unaligned pointer can result in a much slower access time or a machine exception.

A pointer to `void` can be converted to or from a pointer to any incomplete or object type. If a pointer to any incomplete or object type is converted to a pointer to `void` and back, the result compares equal to the original pointer.

An integral constant expression equal to 0, or such an expression cast to the `void *` type, is called a *null pointer constant*. If a null pointer constant is assigned to or compared for equality with a pointer, the constant is converted to a pointer of that type. Such a pointer is called a *null pointer*, and is guaranteed to compare unequal to a pointer to any object or function.

An array designator is automatically converted to a pointer to the array type, and the pointer points to the first element of the array.

6.11.3 Function Argument Conversions

The data types of function arguments are assumed to match the types of the formal parameters unless a function prototype declaration is present. In the presence of a function prototype, all arguments in the function invocation are compared for assignment compatibility to all parameters declared in the function prototype declaration. If the type of the argument does not match the type of the parameter but is assignment compatible, C converts the argument to the type of the parameter (see Section 6.11.1). If an argument in the function invocation is not assignment compatible to a parameter declared in the function prototype declaration, an error message is generated.

If a function prototype is not present, all arguments of type `float` are converted to `double`, all arguments of type `char` or `short` are converted to type `int`, all arguments of type `unsigned char` and `unsigned short` are converted to `unsigned int`, and an array or function name is converted to the address of the named array or function. The compiler performs no other conversions automatically, and any mismatches after these conversions are programming errors.

A *function designator* is an expression that has function type. Except when it is the operand of the `sizeof` operator or the unary & operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*."

7
Statements

This section describes the following kinds of statements in the C programming language. Except as indicated in this chapter, statements are executed in the sequence in which they appear in a function body:

- Labeled statements (Section 7.1)
- Compound statements (Section 7.2)
- Expression statements (Section 7.3)
- Null statements (Section 7.4)
- Selection statements (Section 7.5)
- Iteration statements (Section 7.6)
- Jump statements (Section 7.7)

7.1 Labeled Statements

A label is an identifier used to flag a location in a program as the target of a `goto` statement or `switch` statement. A label has the following syntax:

identifier : *statement*

case *constant-expression : statement*
default *: statement*

The scope of the label is the containing function body. Variables can have the same name as a label in the function because labels and variables have different name spaces (see Section 2.15).

There are three kinds of labeled statements in C:

- Any statement preceded by a label
- A `case` statement
- A `default` statement

The last two statements are discussed in Section 7.5.2 because they can appear only within a `switch` statement.

7.2 Compound Statements

A compound statement, or *block*, allows a sequence of statements to be treated as a single statement. A compound statement begins with a left brace, contains any mix of declarations and statements, and ends with a right brace, as shown in the following example:

```
{
   int a;
   a = 1;
   int b;
   b = 2;
}
```

Note

The ability to mix declarations and statements in any sequence in a compound statement is not allowed in common C, VAX C, and Strict ANSI89 modes. In these modes, the declarations must be specified first, followed by the statements.

Block declarations are local to the block, and, for the rest of the block, they supersede other declarations of the same name in outer scopes.

A block is entered *normally* when control flows into it, or when a `goto` statement transfers control to a label at the beginning of the block itself. Each time the block is entered normally, storage is allocated for `auto` or `register` variables. If, on the other hand, a `goto` statement transfers control to a label *inside* the block or if the block is the body of a `switch` statement, these storage allocations do not occur. For more information about storage classes, see Section 2.10.

Function definitions contain compound statements. The compound statement following the parameter declarations in a function definition is called the function body.

7.3 Expression Statements

Any valid expression can be used as a statement by following the expression with a semicolon, as shown in the following example:

```
i++;
```

This statement increments the value of the variable `i`. Note that `i++` is a valid C expression that can appear in more complex C statements. For more information about the C expressions, see Chapter 6.

7.4 Null Statements

A null statement is used to provide a null operation in situations where the grammar of the language requires a statement, but the program requires no work to be done. The null statement consists of a semicolon:

```
;
```

The null statement is useful with the `if`, `while`, `do`, and `for` statements. The most common use of this statement is in loop operations in which all the loop activity is performed by the test portion of the loop. For example, the following statement finds the first element of an array that has a value of 0:

```
for (i=0; array[i] != 0; i++)
   ;
```

In this example, the `for` statement is executed for its side effects only; the loop body is a null statement. See Section 7.6 for more information about iteration statements.

The null statement is also useful where a label is needed just before a brace that terminates a compound statement. (A label cannot immediately precede the right brace; it must always be attached to a statement.) For example:

```
if (expression1)
{
  ...
  goto label_1;  /* Terminates this part of the if statement */
  ...
label_1: ;
}
else ...
```

7.5 Selection Statements

A selection statement selects among a set of statements depending on the value of a controlling expression. The selection statements are the `if` statement and the `switch` statement, which are discussed in the following sections.

7.5.1 The if Statement

The `if` statement has the following syntax:

if (*expression*)

statement

else$_{opt}$

else-statement$_{opt}$

The statement following the control expression is executed if the value of the control expression is true (nonzero). An `if` statement can be written with an optional `else` clause that is executed if the control expression is false (0).

Consider the following example:

```
if (i < 1)
   funct(i);
else
   {
   i = x++;
   funct(i);
   }
```

In this example, if the value of `i` is less than 1, then the statement `funct(i)` is executed and the compound statement following the keyword `else` is not executed. If the value of `i` is not less than 1, then only the compound statement following the keyword `else` is executed.

The control expression in a selection statement is usually a logical expression, but it can be any expression of scalar type.

When `if` statements are nested, an `else` clause matches the most recent `if` statement that does not have an `else` clause, and is in the same block. For example:

```
if (i < 1)
    {
    if (j < 1)
        funct(j);
    if (k < 1)            /* This if statement is associated with */
        funct(k);
    else                  /* this else clause.                    */
        funct(j + k);
    }
```

7.5.2 The switch Statement

The `switch` statement executes one or more of a series of cases, based on the value of a controlling expression. The `switch` statement has the following syntax:

switch (*expression*)

statement

The usual arithmetic conversions are performed on the control expression, but the result must have an integral type. For more information about data-type conversion, see Section 6.11. The `switch` statement is typically a compound statement, within which are one or more `case` statements executed if the control expression matches the `case`. The syntax for a `case` label and expression follows:

case *constant-expression* : *statement*

The constant expression must have an integral type. No two `case` labels can specify the same value. There is no limit on the number of `case` labels in a `switch` statement.

Only one statement in the compound statement can have the following label:

default :

The `case` and `default` labels can occur in any order, but it is common practice for the `default` statement to follow the `case` statements. Note that execution flows from the selected case into the cases following unless explicit action is taken, such as a `break` statement.

When the `switch` statement is executed, the following sequence takes place:

1. The `switch` control expression is evaluated (and integral promotions applied) and compared with the constant expressions in the `case` labels.

2. If the control expression's value matches a `case` label, control transfers to the statement following that label. If a `break` statement is encountered, the `switch` statement terminates; otherwise, execution continues into the following `case` or default statements until a `break` statement or the end of the `switch` statement is encountered (see Example 7–1).

   A `switch` statement can also be terminated by a `return` or `goto` statement. If a `switch` statement is inside a loop, the `switch` statement is terminated if a `continue` statement terminates the loop. See Section 7.7 for more information about these statements.

3. If the control expression's value does not match any `case` label, and there is a `default` label, control is transferred to the statement following that label. If a `break` statement does not end the `default` statement, and a `case` label follows, that `case` statement is executed.

4. If the control expression's value does not match any `case` label and there is no `default` label, execution of the `switch` statement terminates.

Example 7–1 uses the `switch` statement to count blanks, tabs, and new-line characters entered from the terminal.

Example 7–1 Using switch to Count Blanks, Tabs, and New Lines

```
/*  This program counts blanks, tabs, and new lines in text *
 *  entered from the keyboard.                              */
```

(continued on next page)

Example 7–1 (Cont.) Using switch to Count Blanks, Tabs, and New Lines

```
#include <stdio.h>
main()
{
   int number_tabs = 0, number_lines = 0, number_blanks = 0;
   int ch;
   while ((ch = getchar()) != EOF)
      switch (ch)
         {
❶           case '\t':  ++number_tabs;
❷                       break;
            case '\n':  ++number_lines;
                        break;
            case ' ' :  ++number_blanks;
                        break;
            default:;
         }
   printf("Blanks\tTabs\tNewlines\n");
   printf("%6d\t%6d\t%6d\n", number_blanks,
                             number_tabs,number_lines);
}
```

Key to Example 7–1:

❶ A series of `case` statements is used to increment separate counters depending on the character encountered.

❷ The `break` statement causes control to return to the `while` loop. Control is passed to the `while` loop if the value of `ch` does not match any of the `case` constant expressions.

Without the `break` statements, each `case` would drop through to the next.

If variable declarations appear in the compound statement within a `switch` statement, initializers on `auto` or `register` declarations are ineffective. However, initializations within the statements following a `case` are effective. Consider the following example:

```
switch (ch)
   {
      int nx = 1;          /* Initialization ignored           */
      printf("%d", n);     /* This first printf is not executed */
      case 'a' :
       { int n = 5;        /* Proper initialization occurs     */
         printf("%d", n);
         break; }
      case 'b' :
         { break; }
      default :
         { break; }
   }
```

In this example, if `ch == 'a'`, then the program prints the value 5. If the variable equals any other letter, the program prints nothing because the initialization occurs outside of the `case` label, and statements outside of the `case` label are ineffective.

7.6 Iteration Statements

An iteration statement, or *loop*, repeatedly executes a statement, known as the *loop body*, until the controlling expression is false (0). The control expression must have a scalar type.

The `while` statement evaluates the control expression before executing the loop body (see Section 7.6.1).

The `do` statement evaluates the control expression after executing the loop body; at least one execution of the loop body is guaranteed (see Section 7.6.2).

The `for` statement executes the loop body based on the evaluation of the second of three expressions (see Section 7.6.3).

7.6.1 The while Statement

The `while` statement evaluates a control expression before each execution of the loop body. If the control expression is true (nonzero), the loop body is executed. If the control expression is false (0), the `while` statement terminates. The `while` statement has the following syntax:

while (*expression*)

statement

Consider the following `while` statement:

```
n = 0;
while (n < 10)
   {
      a[n] = n;
      n++;
   }
```

This statement tests the value of `n`; if `n` is less than 10, it assigns `n` to the *n*th element of the array `a` and then increments `n`. The control expression (in parentheses) is then evaluated; if true (nonzero), the loop body is executed again; if false (0), the `while` statement terminates. If the statement `n++` were missing from the loop body, this `while` statement would never terminate. If the statement `n = 0` were replaced by the statement `n = 10`, the control expression is initially false (0), and the loop body is never executed.

7.6.2 The do Statement

The `do` statement evaluates the control expression after each execution of the loop body. The `do` statement has the following syntax:

do

statement
while (*expression*) ;

The loop body is executed at least once. The control expression is evaluated after each execution of the loop body. If the control expression is true (nonzero), the statement is executed again. If the control expression is false (0), the `do` statement terminates.

7.6.3 The for Statement

The `for` statement evaluates three expressions and executes the loop body until the second controlling expression evaluates to false (0). The `for` statement is useful for executing a loop body a specified number of times. The `for` statement has the following syntax:

for (*expression-1*$_{opt}$;

expression-2$_{opt}$; *expression-3*$_{opt}$)
statement

The `for` statement is equivalent to the following code:

expression-1;

while (*expression-2*)

{

statement
expression-3 ;
}

The `for` statement executes the loop body zero or more times. Semicolons (`;`) are used to separate the control expressions. A `for` statement executes the following steps:

1. *expression-1* is evaluated once before the first iteration of the loop. This expression usually specifies the initial values for variables used in the loop.
2. *expression-2* is any scalar expression that determines whether to terminate the loop. *expression-2* is evaluated before each loop iteration. If the expression is true (nonzero), the loop body is executed. If the expression is false (0), execution of the `for` statement terminates.
3. *expression-3* is evaluated after each iteration.
4. The `for` statement executes until *expression-2* is false (0), or until a jump statement, such as `break` or `goto`, terminates execution of the loop.

Any of the three expressions in a `for` loop can be omitted:

- If *expression-2* is omitted, the test condition is always true; that is, the `while` loop equivalent becomes `while(1)`. This is an infinite loop. For example:

  ```
  for (i = 0;  ;i++)
      statement;
  ```

  Infinite loops can be terminated with a `break`, `return`, or `goto` statement within the loop body.

- If either *expression-1* or *expression-3* is omitted from the `for` statement, the omitted expression is evaluated as a `void` expression and is effectively dropped from the expansion. For example:

  ```
  n = 1;
  for ( ; n < 10; n++)
     func(n);
  ```

  In this example, `n` is initialized before the `for` statement is executed.

In relaxed ANSI C mode, the first clause of the `for` statement can be a declaration whose scope includes the remaining clauses of the `for` header and the entire loop body. This is normally used to declare and initialize a local loop control variable. For example:

```
for (int i=0; i<10; i++)
    printf("%d\n", i);
```

7.7 Jump Statements

Jump statements cause an unconditional jump to another statement elsewhere in the code. They are used primarily to interrupt `switch` statements and loops.

The jump statements are the `goto` statement, the `continue` statement, the `break` statement, and the `return` statement, which are discussed in the following sections.

7.7.1 The goto Statement

The `goto` statement unconditionally transfers program control to a labeled statement, where the label identifier is in the scope of the function containing the `goto` statement. The labeled statement is the next statement executed. The `goto` statement has the following syntax:

goto *identifier*;

Care must be taken when branching into a block by using the `goto` statement, because storage is allocated for automatic variables declared within a block when the block is activated. When a `goto` statement branches into a block, automatic variables declared in the block are not initialized.

7.7.2 The continue Statement

The `continue` statement passes control to the end of the immediately enclosing `while`, `do`, or `for` statement. The `continue` statement has the following syntax:

continue;

The `continue` statement is equivalent to a `goto` statement within an iteration statement that passes control to the end of the loop body. For example, the following two loops are equivalent:

```
while(1)
{
   .
   .
   .
  goto label_1;
   .
   .
   .
  label_1:
   ;
 }
```

```
while(1)
{
   .
   .
   .
 continue;
   .
   .
   .

   ;
 }
```

The `continue` statement can be used only in loops. A `continue` inside a `switch` statement that is inside a loop causes continued execution of the enclosing loop after exiting from the body of the `switch` statement.

7.7.3 The break Statement

The `break` statement terminates execution of the immediately enclosing `while`, `do`, `for`, or `switch` statement. Control passes to the statement following the loop body (or the compound statement of a `switch` statement). The `break` statement has the following syntax:

break;

See Example 7–1 which uses a `break` statement within a `switch` statement.

7.7.4 The return Statement

The `return` statement terminates execution of a function and returns control to the calling function, with or without a return value. A function may contain any number of `return` statements. The `return` statement has the following syntax:

return *expression*$_{opt}$;

If present, the expression is evaluated and its value is returned to the calling function. If necessary, its value is converted to the declared type of the containing function's return value.

A `return` statement with an expression cannot appear in a function whose return type is `void`. For more information about the `void` data type and function return types, see Sections 3.5 and 3.4.1.

If there is no expression and the function is not defined as `void`, the return value is undefined. For example, the following `main` function returns an unpredictable value to the operating system:

```
main ( )
  {
   return;
  }
```

Reaching the closing brace that terminates a function is equivalent to executing a `return` statement without an expression.

8

Preprocessor Directives and Predefined Macros

The C preprocessor provides the ability to perform macro substitution, conditional compilation, and inclusion of named files. Preprocessor directives, lines beginning with # and possibly preceded by white space, are used to communicate with the preprocessor.

The following sections describe the preprocessor directives and operators available with the HP C compiler:

- The `#define` and `#undef` directives, and the `#` and `##` operators (Section 8.1)
- The `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` directives, and the `defined` operator (Section 8.2)
- The `#include` directive (Section 8.3)
- The `#line` directive (Section 8.4)
- The `#pragma` directive (Section 8.5)
- The `#error` directive (Section 8.6)
- The null directive (#) (Section 8.7)

Preprocessor directives are independent of the usual scope rules; they remain in effect from their occurrence until the end of the compilation unit or until their effect is canceled.

See Section 8.2 for more information about conditional compilation. See your platform-specific HP C documentation for implementation-defined information about preprocessor directives.

The ANSI standard allows only comments as text following a preprocessing directive. The HP C compiler issues a warning if this syntax rule is violated in all modes but the strict ANSI mode, in which it issues an error message.

8.1 Macro Definition (#define and #undef)

The `#define` directive specifies a macro identifier and a replacement list, and terminates with a new-line character. The replacement list, a sequence of preprocessing tokens, is substituted for every subsequent occurrence of that macro identifier in the program text, unless the identifier occurs inside a character constant, a comment, or a literal string. The `#undef` directive is used to cancel a definition for a macro.

A macro definition is independent of block structure, and is in effect from the `#define` directive that defines it until either a corresponding `#undef` directive or the end of the compilation unit is encountered.

The `#define` directive has the following syntax:

#define *identifier replacement-list newline*

#define *identifier* (*identifier-list*$_{opt}$) *replacement-list newline*

#define *identifier* (. . .) *replacement-list newline*

#define *identifier* (*identifier-list, . . . replacement-list newline*

If the *replacement-list* is empty, subsequent occurrences of the identifier are deleted from the source file.

The first form of the `#define` directive is called an *object-like macro*. The last three forms are called *function-like macros*.

The `#undef` directive has the following syntax:

#undef *identifier newline*

This directive cancels a previous definition of the identifier by `#define`. Redefining a macro previously defined is not legal, unless the new definition is precisely the same as the old.

The replacement list in the macro definition, as well as arguments in a function-like macro reference, can contain other macro references. HP C does not limit the depth to which such references can be nested.

For a given macro definition, any macro names contained in the replacement list are themselves replaced by their currently specified replacement lists. If a macro name being defined is contained in its own replacement list or in a nested replacement list, it is not replaced. These nonreplaced macro names are then no longer available for further replacement, even if they are later examined in contexts in which they would otherwise be replaced.

The following example shows nested #define directives:

```
/*  Show multiple substitutions and listing format. */

#define  AUTHOR  james + LAST

main()
{
   int writer,james,michener,joyce;

   #define LAST michener
   writer = AUTHOR;
   #undef LAST
   #define LAST joyce
   writer = AUTHOR;
}
```

After this example is compiled with the appropriate options to show intermediate macro expansions, the following listing results:

```
    1            /* Show multiple substitutions and listing format. */
    2
    3            #define AUTHOR james + LAST
    4
    5            main()
    6            {
    7              int writer, james, michener, joyce;
    8
    9              #define LAST michener
   10              writer = AUTHOR;
   10.1                      james + LAST
   10.2                      michener
   11             #undef LAST
   12             #define LAST joyce
   13             writer = AUTHOR;
   13.1                      james + LAST
   13.2                      joyce
   14           }
```

On the first pass, the compiler replaces the identifier AUTHOR with the replacement list james + LAST. On the second pass, the compiler replaces the identifier LAST with its currently defined replacement list value. At line 9, the replacement list value for LAST is the identifier michener, so michener is substituted at line 10. At line 12, the replacement list value for LAST is redefined to be the identifier joyce, so joyce is substituted at line 13.

The #define directive may be continued onto subsequent lines if necessary. To do this, end each line to be continued with a backslash (\) immediately followed by a new-line character. The backslash and new-line characters do not become part of the definition. The first character in the next line is logically adjacent to the character that immediately precedes the backslash.

The backslash/newline as a continuation sequence is valid anywhere. However, comments within the definition line can be continued without the backslash/newline.

If you plan to port programs to and from other C implementations, take care in choosing which macro definitions to use within your programs, because some implementations define different macros than others.

8.1.1 Object-Like Form

A preprocessing directive of the following form defines an object-like macro that causes each subsequent occurrence of the macro name to be replaced by the replacement list:

`#define` *identifier replacement-list newline*

An object like macro may be redefined by another `#define` directive provided that the second definition is an object-like macro definition and the two replacement lists are identical. This means that two files, each with a definition of a certain macro, must be consistent in that definition.

The object-like form of macro definition defines a descriptive name for a frequently used token. A common use of the directive is to define the end-of-file (`EOF`) indicator as follows:

```
#define  EOF  (-1)
```

8.1.2 Function-Like Form

The function-like form of macro definition includes a list of parameters. References to such macros look like function calls. When a function is called, control passes from the program to the function at run time; when a macro is referenced, source code is inserted into the program at compile time. The parameters are replaced by the corresponding arguments, and the text is inserted into the program stream.

If there is a ... in the *identifier-list* in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the variable arguments. The number of arguments so combined is such that, following the merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the ...).

An identifier __VA_ARGS__ that occurs in the *replacement-list* of a function-like macro that uses ellipsis notation in the arguments is treated as if it were a parameter, and the variable arguments form the preprocessing tokens used to replace it.

If the replacement list is omitted from the macro definition, the entire macro reference disappears from the source text.

The library macro _toupper, available on some systems in the ctype.h header file, is a good example of macro replacement. This macro is defined as follows:

```
#define  _toupper(c)  ((c) >= 'a'  &&  (c) <= 'z'  ? (c) & 0X5F : (c))
```

When the macro _toupper is referenced, the compiler replaces the macro and its parameter with the replacement list from the directive, substituting the argument of the macro reference for each occurrence of the parameter (c in this case) in the replacement list.

The replacement list of C source code can be translated in the following manner: if parameter c is a lowercase letter (between 'a' and 'z'), the expression evaluates to an uppercase letter (c & 0X5F); otherwise, it evaluates to the character as specified. This replacement list uses the if-then-else conditional operator (?:). For more information about the conditional operator, see Section 6.6. For more information about the bitwise operators, see Section 6.5.6.

8.1.2.1 Rules for Specifying Macro Definitions

Preprocessor directives and macro references have syntax that is independent of the C language. Follow these rules when specifying macro definitions:

- The macro name and the formal parameters are identifiers and are specified according to the rules for identifiers in the C language.
- Spaces, tabs, and comments may be used freely within a #define directive anywhere that the delta symbol (Δ) appears in the following example:

  Δ #Δ define Δ *name*(Δ *parm1*Δ ,Δ *parm2*Δ)Δ \

  Δ *token-string*Δ

  Spaces, tabs, and comments are replaced by a single space.
- White space cannot appear between the name and the left parenthesis that introduces the parameter list. White space may appear inside the replacement list. Also, at least one space, tab, or comment must separate *name* from define.
- The identifier __VA_ARGS__ can occur only in the *replacement-list* of a function-like macro that uses the ellipsis notation in the arguments.

8.1.2.2 Rules for Specifying Macro References

Follow these rules when specifying macro references:

- Comments and white-space characters (spaces, horizontal and vertical tabs, new-line characters, and form feeds) may be used freely within a macro reference anywhere that the delta symbol (Δ) appears in the following example:

  Δ *name*Δ (Δ *arg1*Δ ,Δ *arg2*Δ)

- Arguments consist of arbitrary text. Syntactically, they are not restricted to C expressions. They may contain embedded comments and white space. Comments are replaced with a single space. White space (except for leading and trailing white space) is preserved during the substitution.
- The number of arguments in the reference must match the number of parameters in the macro definition. Null arguments result in undefined behavior.
- Commas separate arguments except where the commas occur inside string or character constants, comments, or pairs of parentheses. Parentheses must be balanced within arguments.

8.1.2.3 Side Effects in Macro Arguments

It is not good programming practice to specify macro arguments that use the increment (++), decrement (−−), and assignment operators (such as +=) or other arguments that can cause side effects. For example, do not pass the following argument to the `_toupper` macro:

```
_toupper(p++)
```

When the argument `p++` is substituted in the macro definition, the effect within the program stream is as follows:

```
((p++) >= 'a' && (p++) <= 'z' ? (p++) & 0X5F : (p++))
```

Because `p` is being incremented, it does not have the same value for each occurrence in this macro replacement. Even if you are aware of possible side effects, the replacement lists within macro definitions can be changed, which changes the side effects without warning.

8.1.3 Conversions to String Literals (#)

The # preprocessor operator is used to convert the argument that follows it to a string literal. The preprocessor operator # can be used only in a function-like macro definition. For example:

```
#include <stdio.h>

#define PR(id) printf("The value of " #id " is %d\n", id)

main()
{
 int i = 10;

 PR(i);
}
```

The output produced is:

```
The value of i is 10
```

The macro call expands in the following steps:

```
/*1*/  printf("The value of " #id " is %d\n", id)
/*2*/  printf("The value of " "i" " is %d\n", 10)
/*3*/  printf("The value of i is %d\n", 10)
```

The unary # operator produces a string from its operand. This example also uses the fact that adjacent string literals are concatenated. If the operand to # contains double quotes or escape sequences, they are also expanded. For example:

```
#include <stdio.h>

#define M(arg) printf(#arg " is %s\n", arg)

main()
{
 M("a\nb\tc");
}
```

The macro call expands using the following steps:

```
/*1*/  printf(#arg " is %s\n", arg)
/*2*/  printf("\"a\\nb\\tc\"" " is %s\n", "a\nb\tc");
/*3*/  printf("\"a\\nb\\tc\" is %s\n", "a\nb\tc");
```

8.1.4 Token Concatenation(##)

The ## preprocessor operator is used to concatenate two tokens into a third valid token, as in the following example:

```
#define glue(a,b) a ## b

main()
{
int wholenum = 5000;

printf("%d", glue(whole,num));
}
```

The preprocessor converts the line `printf("%d", glue(whole,num));` into `printf("%d", wholenum);`, and when executed, the program prints 5000. If the result is not a valid token, an error occurs when the tokens are concatenated.

In HP C, the ## operator is evaluated before any # operators on the line. ## and # operators group left-to-right.

8.2 Conditional Compilation (#if, #ifdef, #ifndef, #else, #elif, #endif, and defined)

Six directives are available to control conditional compilation. They delimit blocks of program text that are compiled only if a specified condition is true. These directives can be nested. The program text within the blocks is arbitrary and may consist of preprocessor directives, C statements, and so on. The beginning of the block of program text is marked by one of three directives:

- `#if`
- `#ifdef`
- `#ifndef`

Optionally, an alternative block of text can be set aside with one of two directives:

- `#else`
- `#elif`

The end of the block or alternative block is marked by the `#endif` directive.

If the condition checked by `#if`, `#ifdef`, or `#ifndef` is true (nonzero), then all lines between the matching `#else` (or `#elif`) and an `#endif` directive, if present, are ignored.

If the condition is false (0), then the lines between the `#if`, `#ifdef`, or `#ifndef` and an `#else`, `#elif`, or `#endif` directive are ignored.

8.2.1 The #if Directive

The `#if` directive has the following syntax:

#if *constant-expression newline*

This directive checks whether the *constant-expression* is true (nonzero). The operand must be a constant integer expression that does not contain any increment (++), decrement (−−), `sizeof`, pointer (*), address (&), and cast operators.

Identifiers in the constant expression either are or are not macro names. There are no keywords, enumeration constants, and so on. The constant expression can also include the `defined` preprocessing operator (see Section 8.2.7).

The constant expression in an `#if` directive is subject to text replacement and can contain references to identifiers defined in previous `#define` directives. The replacement occurs before the expression is evaluated. Each preprocessing token that remains after all macro replacements have occurred is in the lexical form of a token.

If an identifier used in the expression is not currently defined, the compiler treats the identifier as though it were the constant zero.

8.2.2 The #ifdef Directive

The `#ifdef` directive has the following syntax:

#ifdef *identifier newline*

This directive checks whether the identifier is currently defined. Identifiers can be defined by a `#define` directive or on the command line. If such identifiers have not been subsequently undefined, they are considered currently defined.

8.2.3 The #ifndef Directive

The `#ifndef` directive has the following syntax:

#ifndef *identifier newline*

This directive checks to see if the identifier is not currently defined.

8.2.4 The #else Directive

The `#else` directive has the following syntax:

#else *newline*

This directive delimits alternative source text to be compiled if the condition tested for in the corresponding `#if`, `#ifdef`, or `#ifndef` directive is false. An `#else` directive is optional.

8.2.5 The #elif Directive

The `#elif` directive has the following syntax:

#elif *constant-expression newline*

The `#elif` directive performs a task similar to the combined use of the `else-if` statements in C. This directive delimits alternative source lines to be compiled if the constant expression in the corresponding `#if`, `#ifdef`, `#ifndef`, or another `#elif` directive is false and if the additional constant expression presented in the `#elif` line is true. An `#elif` directive is optional.

8.2.6 The #endif Directive

The `#endif` directive has the following syntax:

#endif *newline*

This directive ends the scope of the `#if`, `#ifdef`, `#ifndef`, `#else`, or `#elif` directive.

The number of necessary `#endif` directives changes according to whether the `elif` or `#else` directive is used. Consider the following equivalent examples:

```
#if true
.
.
.
#elif true
.
.
.
#endif
```

```
#if true
.
.
.
.
#else
#if false
.
.
.
#endif
#endif
```

8.2.7 The defined Operator

Another way to verify that a macro is defined is to use the `defined` unary operator. The `defined` operator has one of the following forms:

defined *name*

defined (*name*)

An expression of this form evaluates to 1 if *name* is defined and to 0 if it is not.

The `defined` operator is especially useful for checking many macros with just a single use of the `#if` directive. In this way, you can check for macro definitions in one concise line without having to use many `#ifdef` or `#ifndef` directives.

For example, consider the following macro checks:

```
#ifdef  macro1
printf( "Hello!\n" );
#endif

#ifndef  macro2
printf( "Hello!\n" );
#endif

#ifdef  macro3
printf( "Hello!\n" );
#endif
```

Another use of the `defined` operator is in a single `#if` directive to perform similar macro checks:

```
#if  defined (macro1)  || !defined (macro2) || defined (macro3)
printf( "Hello!\n" );
#endif
```

Note that `defined` operators can be combined in any logical expression using the C logical operators. However, `defined` can only be used in the evaluated expression of an `#if` or `#elif` preprocessor directive.

8.3 File Inclusion (#include)

The `#include` directive inserts the contents of a specified file into the text stream delivered to the compiler. Usually, standard headers and global definitions are included in the program stream with the `#include` directive. This directive has two forms:

#include "*filename*" *newline*

#include <*filename*> *newline*

The format of *filename* is platform-dependent. If the *filename* is enclosed in quotation marks, the search for the named file begins in the directory where the file containing the `#include` directive resides. If the file is not found there, or if the file name is enclosed in angle brackets (< >), the file search follows platform-defined search rules. In general, the quoted form of `#include` is used to include files written by users, while the bracketed form is used to include standard library files.

See your platform-specific HP C documentation for information on the search path rules used for file inclusion.

Macro substitution is allowed within the `#include` preprocessor directive.

For example, the following two directives can be used to include a file:

```
#define  macro1  "file.ext"
#include macro1
```

Defined macros used in `#include` directives must evaluate to one of the two following acceptable `#include` file specifications or an error is reported:

"*filename*"

<filename>

An included file may itself contain `#include` directives. Although the HP C compiler imposes no inherent limitation on the nesting level of inclusion, the permitted depth depends on hardware and operating system restrictions.

8.4 Explicit Line Numbering (#line)

The compiler keeps track of information about line numbers in each file involved in the compilation, and uses the line number when issuing diagnostic messages to the terminal or, when compiling in batch mode, to a log file.

The `#line` directive can be used to alter the line numbers assigned to source code. This directive gives a new line number to the following line, which is then incremented to derive the line number for subsequent lines. The directive can also specify a new file specification for the program source file. The `#line` directive does not change the line numbers in your compilation listing, only the line numbers given in diagnostic messages sent to the terminal screen or log file. This directive is useful for referring to original source files that are preprocessed into C code.

The `#line` directive has three forms:

#line *integer-constant newline*

#line *integer-constant* "*filename*" *newline*

#line *pp-tokens newline*

In the first two forms, the compiler gives the line following a `#line` directive the number specified by the integer constant. The optional *filename* in quotation marks indicates the name of the source file that the compiler will provide in its diagnostic messages. If the file name is omitted, the file name used is the name of the current source file or the last file name specified in a previous `#line` directive.

In the third form, macros in the `#line` directive are expanded before it is interpreted. This allows a macro call to expand into the *integer-constant*, *filename*, or both. The resulting `#line` directive must match one of the other two forms, and is then processed as appropriate.

8.5 Implementation-Specific Preprocessor Directive (#pragma)

The `#pragma` directive is a standard method for implementing platform-dependent features. This directive has the following syntax:

#pragma *pp-tokens*$_{opt}$ *newline*

The supported pragmas vary across platforms. All unrecognized pragmas are diagnosed with an informational message. See your platform-specific HP C documentation for a list of supported pragmas.

Some pragma directives are subject to macro expansion. They are:

```
builtins            inline                linkage       standard
dictionary          noinline              module        nostandard
extern_model        member_alignment      message       use_linkage
extern_prefix       nomember_alignment
```

The following pragmas are also subject to macro expansion, primarily for use in preprocess-only mode (that is, with the /PREPROCESS_ONLY qualifier on OpenVMS systems or the -E switch on Tru64 UNIX systems), and are not normally used when generating an object module with the HP C compiler:

- `_KAP`—Relevant only to the KAPC product.
- `define_template`—Relevant only to HP C++.
- `code_psect`
- `linkage_psect`

Note

An _nm suffix can be appended to any of the above-listed macros to prevent macro expansion. For example, to prevent macro expansion on #pragma inline, specify it as #pragma inline_nm.

Also, to provide macro-expansion support to those pragmas not listed above, all pragmas (including those that are already specified as undergoing macro expansion) have a *pragma-name*_m version, which makes the pragma subject to macro expansion. For example, #pragma assert is not subject to macro expansion, but #pragma assert_m is.

A macro reference can occur anywhere after the keyword pragma. For example:

```
#define opt inline
#define f func
#pragma opt(f)
```

After both macros are expanded, the #pragma directive becomes #pragma inline (func).

Note

Macro expansion is a feature of pragmas introduced in early versions of DEC C and is retained for backward compatibility.

Pragmas added in more recent versions of the compiler and pragmas added in the future have changed that practice to conform to the defacto industry standard of not performing macro expansion. (ANSI C places no requirement on macro expansion of pragmas.)

The following describes how the compiler decides whether or not to macro-expand a given pragma:

In compilation modes other than /STANDARD=COMMON OpenVMS systems) or -std0 (Tru64 UNIX systems), do Step 1:

Step 1:

The token following the keyword pragma is first checked to see if it is a currently-defined macro. If it is a macro and the identifier does not match the name of a pragma that is not subject to macro expansion, then just that macro (with its arguments, if function-like) is expanded. The tokens

produced by that macro expansion are then processed along with the rest of the tokens on the line in Step 2.

In all compilation modes, do Step 2:

Step 2:

The first token following the keyword `pragma` is checked to see if it matches the name of a pragma that is subject to macro expansion. If it does, then macro expansion is applied to that token and to the rest of tokens on the line.

The test for matching a known pragma permits an optional double leading underscore. For example, `#pragma __nostandard` is equivalent to `#pragma standard`.

Example

The following example illustrates that for pragmas coded directly with a name that matches a known pragma, the macro-expansion behavior is generally the same in all modes and is backward-compatible. It is only in cases where a pragma was coded with a name that was not the name of a known pragma, expecting macro expansion to produce the pragma name, that backward-compatibility is broken, and then only in common mode. The exception is made in common mode to maintain compatibility with the Tru64 UNIX preprocessor.

```
#define pointer_size error
#define m1 e1
#define e1 pointer_size 32
#define standard message
#define x disable(all)
#define disable(y) enable(y)

#pragma pointer_size 32  /* In common mode, Step 1 skipped.
                            In other modes, Step 1 finds that pointer_size
                                 is known not to expand.
                            In any mode, Step 2 finds pointer_size is
                                 not a pragma requiring expansion. */

#pragma m1   /* In common mode, Step 1 skipped.
                In other modes, Step 1 expands m1 to pointer_size 32.
                In common mode, Step 2 finds m1 is not a pragma requiring
                       expansion.
                In other modes, Step 2 finds pointer_size is not a pragma
                       requiring expansion. */

#pragma standard x  /* In common mode, Step 1 skipped.
                       In other modes, Step 1 expands to message x.
                       In common mode, Step 2 expands to message enable(all).
                       In other modes, Step 2 expands message x to
                          message enable(all). */
```

8.6 Error Directive (#error)

The `#error` preprocessor directive issues an E-level diagnostic message and continues compilation, but no object module is produced. This directive has the following syntax:

#error $message_{opt}$ *newline*

8.7 Null Directive (#)

A preprocessing directive of the form # *newline* is a null directive and has no effect.

8.8 Predefined Macro Names

The following sections describe the predefined macro names that are provided to assist in transporting code and performing simple tasks common to many programs.

8.8.1 The __DATE__ Macro

The __DATE__ macro evaluates to a string literal specifying the date on which the compilation started. The date has the following format:

"Mmm dd yyyy"

The names of the months are the same as those generated by the `asctime` library function. The first d is a space if dd is less than 10. For example:

```
printf("%s",_ _DATE_ _);
```

The value of this macro remains constant throughout the translation unit.

8.8.2 The __FILE__ Macro

The __FILE__ macro evaluates to a string literal specifying the file specification of the current source file. For example:

```
printf("file %s", _ _FILE_ _);
```

8.8.3 The __LINE__ Macro

The __LINE__ macro evaluates to a decimal constant specifying the number of the line in the source file containing the macro reference. For example:

```
printf("At line %d in file %s", _ _LINE_ _, _ _FILE_ _);
```

8.8.4 The __TIME__ Macro

The __TIME__ macro evaluates to a string specifying the time that the compilation started. The time has the following format (the same as the `asctime` function):

hh:mm:ss

For example:

```
printf("%s", __TIME__);
```

The value of this macro remains constant throughout the translation unit.

8.8.5 The __STDC__, __STDC_HOSTED__, and __STDC_VERSION Macros

The __STDC__ macro is set to 1 to indicate a conforming implementation.

The __STDC_HOSTED__ macro is set to 1 if the implementation is a hosted implementation. Otherwise, the macro is not defined.

The __STDC_VERSION__ macro is set to 199901L in the default mode of the HP C compiler.

The compiler mode being used (as indicated by the /STANDARD qualifier option or `-std` switch) determines whether or not these macros are defined and, if defined, what value they are set to. For additional detail, see your platform-specific HP C documentation.

The value of these macros remains constant throughout the translation unit.

8.8.6 The __STDC_ISO_10646__ Macro

The __STDC_ISO_10646__ macro evaluates to an integer constant of the form *yyyymmL* (for example, 199712L), intended to indicate that values of type `wchar_t` are the coded representations of the characters defined by ISO/IEC 10646, along with all amendments and technical corrigenda as of the specified year and month.

8.8.7 System-Identification Macros

HP C defines platform-specific macros that can be used to identify the system on which the program is running. These macros can assist in writing code that executes conditionally depending on whether the program is running on an HP system or some other system, or one HP C platform or another.

These macro definitions can be used to separate portable and nonportable code in a C program by enclosing the nonportable code in conditionally compiled sections.

They can also be used to conditionally compile sections of C programs used on more than one operating system to take advantage of system-specific features. See Section 8.2 for more information about using the conditional-compilation preprocessor directives.

See your platform-specific HP C documentation for a list of the system-identification macros.

8.9 The __func__ Predeclared Identifier

The __func__ predeclared identifier evaluates to a static array of `char` initialized with the spelling of the function's name. It is visible anywhere within the body of a function definition.

For example, a function defined as follows will print "f1".

```
void f1(void) {printf("%s\n", __func__);}
```

9

The ANSI C Standard Library

The ANSI C standard defines a set of functions, as well as related types and macros, to be provided with any implementation of ANSI C. This chapter lists and briefly describes the ANSI-conformant library features common to all HP C platforms. See your HP C library routine documentation for a detailed description of these routines and their use in your system environment, and for additional headers, functions, types, and macros that may be available on your operating system.

All library functions are declared in a *header* file. To make the contents of a header file available to your program, include the header file with an `#include` preprocessor directive. For example:

```
#include <stddef.h>
```

Each header file declares a set of related functions, as well as defining any types and macros needed for their use.

The standard headers are:

- Diagnostics: `<assert.h>` (Section 9.1)
- Complex Arithmetic: `<complex.h>` (Section 9.2)
- Character processing: `<ctype.h>` (Section 9.3)
- Error codes: `<errno.h>` (Section 9.4)
- ANSI C limits: `<limits.h>` and `<float.h>` (Section 9.5)
- Localization: `<locale.h>` (Section 9.6)
- Mathematics: `<math.h>` (Section 9.7)
- Nonlocal jumps: `<setjmp.h>` (Section 9.8)
- Signal handling: `<signal.h>` (Section 9.9)
- Variable arguments: `<stdarg.h>` (Section 9.10)
- Boolean type and values: `<stdbool.h>` (Section 9.11)

- Common definitions: <stddef.h> (Section 9.12)
- Input/output: <stdio.h> (Section 9.13)
- General utilities: <stdlib.h> (Section 9.14)
- String processing: <string.h> (Section 9.15)
- Type-generic Math: <tgmath.h> (Section 9.16)
- Date and time: <time.h> (Section 9.17)

Header files can be included in any order. Each can be included more than once in a given scope with no effect different from being included once. However, the effect of including <assert.h> depends on the definition of NDEBUG. Include headers outside of any external declaration or definition, and before any reference to the functions, types, or macros declared or defined in the headers. If an identifier is declared or defined in more than one included header, the second and subsequent headers containing that identifier can be included after the initial reference to that identifier.

9.1 Diagnostics (<assert.h>)

The header <assert.h> defines the assert macro and refers to another macro, NDEBUG, defined elsewhere. If NDEBUG is defined as a macro name at the point in the source file where <assert.h> is included, the assert macro is defined as follows:

```
#define assert(ignore) ((void) 0)
```

Macro

`void assert(int expression);`

Puts diagnostics into programs. If *expression* is false (zero), the assert macro writes information about the particular call that failed on the standard error file in an implementation-defined format. It then calls the abort function. The assert macro returns no value.

9.2 Complex Arithmetic(<complex.h>)

The <complex.h> header file defines macros and declares functions that support complex arithmetic.

Each synopsis specifies a family of functions consisting of a principal function with one or more double complex parameters and a double complex or double return value; and other functions with the same name but with f and l suffixes which are corresponding functions with float and long double parameters and return values.

Macros

`complex`

Expands to `_Complex`.

`_Complex_I`

Expands to a constant expression of type `const float _Complex`, with the value of the imaginary unit. The imaginary unit is a number i such that $i^2 = -1$.

`imaginary`
`_Imaginary_I`

Defined if and only if the implementation supports imaginary types; if defined, they expand to `_Imaginary` and a constant expression of type `const float _Imaginary` with the value of the imaginary unit.

`I`

Expands to `_Imaginary_I` or `_Complex_I`. If `_Imaginary_I` is not defined, `I` expands to `_Complex_I`.

Trigonometric Functions

The `cacos` functions

```
#include <complex.h>
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

The `cacos` functions compute the complex arc cosine of z, with branch cuts outside the interval $[-1,+1]$ along the real axis.

The `cacos` functions return the value, in radians, of the complex arc cosine of z in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[0, \pi]$ along the real axis.

The `casin` functions

```
#include <complex.h>
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

The `casin` functions compute the complex arc sine of z, with branch cuts outside the interval $[-1,+1]$ along the real axis.

The `casin` functions return the value, in radians, of the complex arc sine of z in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

The catan functions

```
#include <complex.h>
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

The catan functions compute the complex arc tangent of z, with branch cuts outside the interval [$-$i,+i] along the imaginary axis.

The catan functions return the value, in radians, of the complex arc tangent of z in the range of a strip mathematically unbounded along the imaginary axis and in the interval [$-$ $\pi/2, +\pi/2$] along the real axis.

The ccos functions

```
#include <complex.h>
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

The ccos functions return the complex cosine of z.

The csin functions

```
#include <complex.h>
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

The csin functions return the complex sine of z.

The ctan functions

```
#include <complex.h>
double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

The ctan functions return the complex tangent of z.

Hyperbolic Functions

The cacosh functions

```
#include <complex.h>
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

The cacosh functions compute the complex arc hyperbolic cosine of z, with a branch cut at values less than 1 along the real axis.

The cacosh functions return the value, in radians, of the complex arc hyperbolic cosine of z in the range of a a half-strip of non-negative values along the real axis and in the interval [$-i\pi, +i\pi$] along the imaginary axis.

The casinh functions

```
#include <complex.h>
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

The casinh functions compute the complex arc hyperbolic sine of z, with branch cuts outside the interval [−i,+i] along the imaginary axis.

The casinh functions return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi\ /2, +i\pi/2]$ along the imaginary axis.

The catanh functions

```
#include <complex.h>
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

The catanh functions compute the complex arc hyperbolic tangent of z, with branch cuts outside the interval [−1,+1] along the real axis.

The catanh functions return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

The ccosh functions

```
#include <complex.h>
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

The ccosh functions return the complex hyperbolic cosine of z.

The csinh functions

```
#include <complex.h>
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

The csinh functions return the complex hyperbolic sine of z.

The ctanh functions

```
#include <complex.h>
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

The ctanh functions return the complex hyperbolic tangent of *z*.

Exponential and Logarithmic Functions

The cexp functions

```
#include <complex.h>
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

The cexp functions return the complex base *e* exponential of *z*.

The clog functions

```
#include <complex.h>
double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

The clog functions compute the complex natural (base *e*) logarithm of *z*, with a branch cut along the negative real axis.

The clog functions return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

Power and Absolute-Value Functions

The cabs functions

```
#include <complex.h>
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
```

The cabs functions return the complex absolute value (also called norm, modulus, or magnitude) of *z*.

The cpow functions

```
#include <complex.h>
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x,
                          long double complex y);
```

The cpow functions compute the complex power function xy, with a branch cut for the first parameter along the negative real axis.

The cpow functions return the complex power function value.

The csqrt functions

```
#include <complex.h>
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

The csqrt functions compute the complex square root of z, with a branch cut along the negative real axis.

The csqrt functions return the complex square root value, in the range of the right half-plane (including the imaginary axis).

Manipulation Functions

The carg functions

```
#include <complex.h>
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

The carg functions compute the argument (also called phase angle) of z, with a branch cut along the negative real axis.

The carg functions return the value of the argument in the interval $[-\pi, +\pi]$.

The cimag functions

```
#include <complex.h>
double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

The cimag functions compute the imaginary part of z and return it as a real.

The conj functions

```
#include <complex.h>
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

The conj functions compute complex conjugate of *z*, by reversing the sign of its imaginary part.

The conj functions return the complex conjugate value.

The cproj functions

```
#include <complex.h>
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

The cproj functions compute a projection of *z* onto the Riemann sphere: *z* projects to *z* except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If *z* has an infinite part, then cproj(*z*) is equivalent to:

```
INFINITY + I * copysign(0.0, cimag(z))
```

The cproj functions return the value of the projection onto the Riemann sphere.

The creal functions

```
#include <complex.h>
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

The creal functions compute and return the real part of *z*.

For a variable *z* of complex type, *z* == creal(*z*) + cimag(*z*)*I.

9.3 Character Processing (<ctype.h>)

The <ctype.h> header file declares several functions for testing characters. For each function, the argument is an int whose value must be EOF or representable as an unsigned char, and the return value is an integer.

Functions

`int isalnum(int c);`

Returns a nonzero integer if the character passed to it is an alphanumeric ASCII character. Otherwise, `isalnum` returns 0.

`int isalpha(int c);`

Returns a nonzero integer if the character passed to it is an alphabetic ASCII character. Otherwise, `isalpha` returns 0.

`int iscntrl(int c);`

Returns a nonzero integer if the character passed to it is an ASCII DEL character (177 octal, 0x7F hex) or any nonprinting ASCII character (a code less than 40 octal, 0x20 hex). Otherwise, `iscntrl` returns 0.

`int isdigit(int c);`

Returns a nonzero integer if the character passed to it is a decimal digit character (0 to 9). Otherwise, `isdigit` returns 0.

`int isgraph(int c);`

Returns a nonzero integer if the character passed to it is a graphic ASCII character (any printing character except a space character). Otherwise, `isgraph` returns 0.

`int islower(int c);`

Returns a nonzero integer if the character passed to it is a lowercase alphabetic ASCII character. Otherwise, `islower` returns 0.

`int isprint(int c);`

Returns a nonzero integer if the character passed to it is an ASCII printing character, including a space character. Otherwise, `isprint` returns 0.

`int ispunct(int c);`

Returns a nonzero integer if the character passed to it is an ASCII punctuation character (any printing character that is nonalphanumeric and greater than 40 octal, 0x20 hex). Otherwise, `ispunct` returns 0.

`int isspace(int c);`

Returns a nonzero integer if the character passed to it is white space. Otherwise, `isspace` returns 0. The standard white space characters are:

- space (' ')
- form feed ('\f')

- new line ('\n')
- carriage return ('\r')
- horizontal tab ('\t')
- vertical tab ('\v')

`int isupper(int c);`

Returns a nonzero integer if the character passed to it is an uppercase alphabetic ASCII character. Otherwise, `isupper` returns 0.

`int isxdigit(int c);`

Returns a nonzero integer if the character passed to it is a hexadecimal digit (0 to 9, A to F, or a to f). Otherwise, `isxdigit` returns 0.

`int tolower(int c);`

Converts an uppercase letter to lowercase. `c` remains unchanged if it is not an uppercase letter.

`int toupper(int c);`

Converts a lowercase letter to uppercase. `c` remains unchanged if it is not a lowercase letter.

9.4 Error Codes (<errno.h>)

The `<errno.h>` header file defines several macros used for error reporting.

Macros

`EDOM`
`ERANGE`

Error codes that can be stored in `errno`. They expand to integral constant expressions with unique nonzero values.

Variable or Macro

`errno`

An external variable or a macro that expands to a modifiable lvalue with type `int`, depending on the operating system.

The `errno` variable is used for holding implementation-defined error codes from library routines. All error codes are positive integers. The value of `errno` is 0 at program startup, but is never set to 0 by any library function. Therefore, `errno` should be set to 0 before calling a library function and then inspected afterward.

9.5 ANSI C Limits (<limits.h> and <float.h>)

The <limits.h> and <float.h> header files define several macros that expand to various implementation-specific limits and parameters, most of which describe integer and floating-point properties of the hardware. See your platform-specific HP C documentation for details.

9.6 Localization (<locale.h>)

The <locale.h> header file declares two functions and one type and defines several macros.

Type

struct lconv

A structure containing members relating to the formatting of numeric values. The structure contains the following members in any order, with values shown in the comments:

```
char *decimal_point;        /*  "."       */
char *thousands_sep;        /*  ""        */
char *grouping;             /*  ""        */
char *int_curr_symbol;      /*  ""        */
char *currency_symbol;      /*  ""        */
char *mon_decimal_point;    /*  ""        */
char *mon_thousands_sep;    /*  ""        */
char *mon_grouping;         /*  ""        */
char *positive_sign;        /*  ""        */
char *negative_sign;        /*  ""        */
char int_frac_digits;       /*  CHAR_MAX  */
char frac_digits;           /*  CHAR_MAX  */
char p_cs_precedes;         /*  CHAR_MAX  */
char p_sep_by_space;        /*  CHAR_MAX  */
char n_cs_precedes;         /*  CHAR_MAX  */
char n_sep_by_space;        /*  CHAR_MAX  */
char p_sign_posn;           /*  CHAR_MAX  */
char n_sign_posn;           /*  CHAR_MAX  */
```

These members are described under the localeconv function in this section.

Macros

NULL
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME

Expand to integral constant expressions with distinct values, and can be used as the first argument to the setlocale function.

Functions

```
char *setlocale(int category, const char *locale);
```

Selects the appropriate portion of the program's locale as specified by the *category* and *locale* arguments. This function can be used to change or query the program's entire current locale or portions thereof.

The following values can be specified for the *category* argument:

LC_ALL—affects the program's entire locale.

LC_COLLATE—affects the behavior of the strcoll and strxfrm functions.

LC_CTYPE—affects the behavior of the character-handling functions and multibyte functions.

LC_MONETARY—affects the monetary-formatting information returned by the localeconv function.

LC_NUMERIC—affects the decimal-point character for the formatted I/O functions and string-conversion functions, as well as the nonmonetary formatting information returned by the localeconv function.

LC_TIME—affects the behavior of the strftime function.

The following values can be specified for the *locale* argument:

- "C"—specifies the minimal environment for C translation
- ""—specifies the use of the environment variable corresponding to *category*. If this environment variable is not set, the LANG environment variable is used. If LANG is not set, an error is returned.

At program startup, the equivalent of the following is executed:

```
setlocale(LC_ALL, "C");
```

The setlocale function returns one of the following:

- If a pointer to a string is specified for *locale* and the selection can be honored, setlocale returns a pointer to the string associated with the specified *category* for the new locale. If the selection cannot be honored, setlocale returns a null pointer and the program's locale is not changed.

- If a null pointer is specified for *locale*, `setlocale` returns a pointer to the string associated with the *category* for the program's current locale. The program's locale is not changed.

In either case, the returned pointer to the string is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. This string must not be modified by the program, but it can be overwritten by subsequent calls to `setlocale`.

```
struct lconv *localeconv(void);
```

Sets the components of an object with type `struct lconv` with values appropriate for formatting numeric quantities according to the rules of the current locale.

The structure members with type `char *` are pointers to strings, any of which (except `decimal_point`) can point to "", which indicates that the value has zero length or is not available in the current locale. Structure members of type `char` are nonnegative numbers, any of which can be `CHAR_MAX` to indicate that the value is not available in the current locale. Structure members include the following:

`char *decimal_point`

The decimal-point character used to format nonmonetary quantities.

`char *thousands_sep`

The character used to separate groups of digits before the decimal point in formatted nonmonetary quantities.

`char *grouping`

A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

`char *int_curr_symbol`

The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in *ISO 4217 Codes for the Representation of Currency and Funds*. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

`char *currency_symbol`

The local currency symbol applicable to the current locale.

`char *mon_decimal_point`

The decimal-point character used to format monetary quantities.

`char *mon_thousands_sep`

The character used to separate groups of digits before the decimal point in formatted monetary quantities.

`char *mon_grouping`

A string whose elements indicate the size of each group of digits in formatted monetary quantities.

`char *positive_sign`

The string used to indicate a nonnegative formatted monetary quantity.

`char *negative_sign`

The string used to indicate a negative formatted monetary quantity.

`char int_frac_digits`

The number of fractional digits to be displayed in internationally formatted monetary quantities.

`char frac_digits`

The number of fractional digits to be displayed in formatted monetary quantities.

`char p_cs_precedes`

Set to 1 if the `currency_symbol` precedes the value for a nonnegative formatted monetary quantity; set to 0 if the `currency_symbol` follows the value.

`char p_sep_by_space`

Set to 1 if the `currency_symbol` is separated by a space from the value for a nonnegative formatted monetary quantity; set to 0 if there is no space.

`char n_cs_precedes`

Set to 1 if the `currency_symbol` precedes the value for a negative formatted monetary quantity; set to 0 if the `currency_symbol` follows the value.

`char n_sep_by_space`

Set to 1 if the `currency_symbol` is separated by a space from the value for a negative formatted monetary quantity; set to 0 if there is no space.

`char p_sign_posn`

Set to a value indicating the positioning of the `positive_sign` for a nonnegative formatted monetary quantity.

`char n_sign_posn`

Set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity.

The elements of `grouping` and `mon_grouping` are interpreted according to the following:

- `CHAR_MAX`—no further grouping is to be performed.
- 0—the previous element is to be repeatedly used for the remainder of the digits.
- *other*—the integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The value of `p_sign_posn` and `n_sign_posn` is interpreted as follows:

- 0—parentheses surround the quantity and `currency_symbol`
- 1—the sign string precedes the quantity and `currency_symbol`
- 2—the sign string follows the quantity and `currency_symbol`
- 3—the sign string immediately precedes the `currency_symbol`
- 4—the sign string immediately follows the `currency_symbol`

The localeconv function returns a pointer to the filled in structure. The structure must not be modified by the program, but might be overwritten by subsequent calls to localeconv or to setlocale with categories LC_ALL, LC_MONETARY, or LC_NUMERIC.

9.7 Mathematics (<math.h>)

The <math.h> header file defines types, macros, and several mathematical functions. The functions take double arguments and return double-precision values.

The behavior of the functions in this header is defined for all representable values of their input arguments. Each function executes as if it were a single operation, without generating any externally visible exceptions.

For all functions, a *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any domain errors. On a domain error, the function returns an implementation-defined value; the value of the EDOM macro is stored in errno.

For all functions, a *range error* occurs if the result of the function cannot be represented as a double value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro HUGE_VAL, with the same sign (except for the tan function) as the correct value of the function; the value of the ERANGE macro is stored in errno. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns 0; whether the value of the ERANGE macro is stored in errno is implementation-defined.

Macros

HUGE_VAL

Expands to a positive double expression.

INFINITY

Expands to a constant expression of type float representing positive or unsigned infinity, if available; otherwise, expands to a positive constant of type float that overflows at translation time.

NAN

Expands to a constant expression of type float representing a quiet NaN.

Trigonometric Functions

```
double acos(double x);
```

Returns the value, in radians, of the arc cosine of x in the range $[0, \pi]$. A domain error occurs for arguments not in the interval $[-1,+1]$.

```
double asin(double x);
```

Returns the value, in radians, of the arc sine of x in the range $[-\pi/2, +\pi/2]$. A domain error occurs for arguments not in the interval $[-1,+1]$.

```
double atan(double x);
```

Returns the value, in radians, of the arc tangent of x in the range $[-\pi/2, +\pi/2]$.

```
double atan2(double y, double x);
```

Returns the value, in radians, of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value. The value returned is in the range $[-\pi, +\pi]$. A domain error may occur if both arguments are 0.

```
double cos(double x);
```

Returns the value, in radians, of the cosine of x.

```
double sin(double x);
```

Returns the value, in radians, of the sine of x.

```
double tan(double x);
```

Returns the value, in radians, of the tangent of x.

Hyperbolic Functions

```
double cosh(double x);
```

Returns the value of the hyperbolic cosine of x. A range error occurs if the magnitude of x is too large.

```
double sinh(double x);
```

Returns the value of the hyperbolic sine of x. A range error occurs if the magnitude of x is too large.

```
double tanh(double x);
```

Returns the value of the hyperbolic tangent of x.

Exponential and Logarithmic Functions

`double exp(double x);`

Returns the value of the exponential function of *x*. A range error occurs if the magnitude of *x* is too large.

`double frexp(double value, int *eptr);`

Breaks the floating-point number *value* into a normalized fraction in the interval [1/2, 1) or 0, which it returns, and an integral power of 2, which it stores in the `int` object pointed to by *eptr*. If *value* is 0, both parts of the result are 0.

`double ldexp(double x, int exp);`

Multiplies a floating-point number by an integral power of 2, and returns the value $x \times 2^{exp}$. A range error may occur.

`double log(double x);`

Returns the natural logarithm of *x*. A domain error occurs if the argument is negative. A range error may occur if the argument is 0.

`double log10(double x);`

Returns the base-ten logarithm of *x*. A domain error occurs if *x* is negative. A range error may occur if *x* is 0.

`double modf(double value, double *iptr);`

Breaks the argument *value* into integral and fractional parts, each of which has the same sign as the argument. The `modf` function returns the signed fractional part and stores the integral part as a `double` in the object pointed to by *iptr*.

Power Functions

`double pow(double x, double y);`

Returns the value x^y. A domain error occurs if *x* is negative and *y* is not an integral value. A domain error occurs if the result cannot be represented when *x* is 0 and *y* is less than or equal to 0. A range error may occur.

`double sqrt(double x);`

Returns the nonnegative square root of *x*. A domain error occurs if *x* is negative.

Nearest Integer, Absolute Value, and Remainder Functions

`double ceil(double x);`

Returns the smallest integral value not less than *x*.

`double fabs(double x);`

Returns the absolute value of a floating-point number *x*.

`double floor(double x);`

Returns the largest integral value not greater than *x*.

`double fmod(double x, double y);`

Computes the floating-point remainder of *x*/*y*. The `fmod` function returns the value $x - i * y$, for some integer i such that if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y. The function returns 0 if y is 0.

9.8 Nonlocal Jumps (<setjmp.h>)

The `<setjmp.h>` header file contains declarations that provide a way to avoid the normal function call and return sequence, typically to permit an intermediate return from a nested function call.

Macro

`int setjmp(jmp_buf env)`

Sets up the local `jmp_buf` buffer and initializes it for the jump (the jump itself is performed with `longjmp`.) This macro saves the program's calling environment in the environment buffer specified by the *env* argument for later use by the `longjmp` function. If the return is from a direct invocation, `setjmp` returns 0. If the return is from a call to `longjmp`, `setjmp` returns a nonzero value.

Type

`jmp_buf`

An array type suitable for holding the information needed to restore a calling environment.

Function

`void longjmp(jmp_buf env, int value;)`

Restores the context of the environment buffer *env* that was saved by invocation of the `setjmp` function in the same invocation of the program. The `longjmp` function does not work if called from a nested signal handler; the result is undefined.

The value specified by *value* is passed from `longjmp` to `setjmp`. After `longjmp` is completed, program execution continues as if the corresponding invocation of `setjmp` had just returned *value*. If *value* is passed to `setjmp` as 0, it is converted to 1.

9.9 Signal Handling (<signal.h>)

The `<signal.h>` header file declares a type and two functions and defines several macros for handling exception conditions that might be reported during program execution.

Type

`sig_atomic_t`

The integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

Macros

`SIG_DFL`
`SIG_ERR`
`SIG_IGN`

Expand to constant expressions with distinct values that have a type compatible with the second argument to, and the return value of, the `signal` function, and whose value compares unequal to the address of any declarable function.

Functions

`void (*signal(int sig, void (*handler) (int))) (int);`

Determines how subsequent signals are handled. Signals are handled in the following way:

1. If the value of `handler` is `SIG_DFL`, default handling of that signal occurs.
2. If the value of `handler` is `SIG_IGN`, the signal is ignored.

3. Otherwise, when that signal occurs, a function pointed to by `handler` is called with the argument of the type of signal. Such a function is called a *signal handler*. Valid signals include:

   - SIGABRT—abnormal termination, such as from the `abort` function
   - SIGFPE—arithmetic error, such as zero divide or overflow
   - SIGILL—invalid function image, such as an invalid instruction
   - SIGINT—interactive attention, such as an interrupt
   - SIGSEGV—invalid access to storage, such as outside of memory limit
   - SIGTERM—termination request sent to the program

   Any other signals are operating-system dependent.

   If the request can be honored, the `signal` function returns the value of `handler` for the most recent call to `signal` for the specified signal `sig`. Otherwise, a value of `SIG_ERR` is returned and an implementation-defined positive value is stored in `errno`.

```
int raise(int sig);
```

Sends the signal `sig` to the executing program. The `raise` function returns 0 if successful and nonzero if unsuccessful.

9.10 Variable Arguments (<stdarg.h>)

The `<stdarg.h>` header file declares a type and defines three macros for advancing through a list of function arguments of varying number and type.

Type

`va_list`

A type suitable for holding information needed by the macros `va_start`, `va_arg`, and `va_end`.

To access varying arguments, the called function must declare an object (referred to as *ap* in this section) that has the type `va_list`:

```
va_list ap;
```

The object *ap* can be passed as an argument to another function. If that function invokes the `va_arg` macro with parameter *ap*, the value of *ap* in the calling function is indeterminate and is passed to the `va_end` macro before any further reference to *ap*.

Macros

`void va_start(va_list ap, parmN);`

Initializes *ap* for subsequent use by `va_arg` and `va_end`. The `va_start` macro must be invoked before any access to the unnamed arguments.

The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list of the function definition. If *parmN* is declared with the `register` storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default arguments promotions, the behavior is undefined. The `va_start` macro returns no value.

type `va_arg(va_list ap, type);`

Expands to an expression that has the type and value of the next argument in the call. The parameter *ap* is the same as the `va_list` *ap* that was initialized by `va_start`. Each invocation of `va_arg` modifies *ap* so that the values of successive arguments are returned in turn. The parameter *type* is a type name specified such that the type of a pointer to an object that has the specified type can be obtained by postfixing an asterisk (*) to *type*. The behavior is undefined if there is no actual next argument, or if *type* is not compatible with the type of the next actual argument (as promoted according to the default argument promotions).

The first invocation of `va_arg` after that of `va_start` returns the value of the argument after that specified by *parmN*. Successive invocations return the values of the remaining arguments in turn.

`void va_end(va_list ap);`

Facilitates a normal return from the function whose variable argument list was referred to by the expansion of `va_start` that initialized the `va_list` *ap* object. The `va_end` macro can modify *ap* so that it can no longer be used (without an intervening invocation of `va_start`). If there is no corresponding invocation of `va_start` or if `va_end` is not invoked before the return, the behavior is undefined. The `va_end` macro returns no value.

9.11 Boolean Type and Values(<stdbool.h>)

The `<stdbool.h>` header file defines four macros.

Macros

`bool`

Expands to `_Bool`.

`true`
`false`
`__bool_true_false_are_defined`

Suitable for use in `#if` preprocessing directives.

`true` expands to the integer constant 1.

`false` expands to the integer constant 0.

`__bool_true_false_are_defined` expands to the integer constant 1.

9.12 Common Definitions (<stddef.h>)

The `<stddef.h>` header file defines several types and macros, some of which are also defined in other header files.

Types

`ptrdiff`

A signed integral type of the result of subtracting two pointers.

`size_t`

An unsigned integral type of the result of the `sizeof` operator.

`wchar_t`

An integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

Macros

`NULL`

Expands to an implementation-defined null pointer constant.

`offsetof(`*type*`, `*member-designator*`)`

Expands to an integral constant expression that has type `size_t` and a value that is the offset, in bytes, to the structure member (specified by *member-designator*) from the beginning of its structure (specified by *type*). The *member-designator* is such that the expression `&(t.`*member-designator*`)` evaluates to an address constant given the following:

```
static type t;
```

If the specified member is a bit field, the behavior is undefined.

9.13 Standard Input/Output (<stdio.h>)

The `<stdio.h>` header file declares three types, several macros, and many functions for performing text input and output. A text stream consists of a sequence of lines; each line ends with a new-line character.

Types

`size_t`

An unsigned integral type of the result of the `sizeof` operator.

`FILE`

An object type capable of recording all the information needed to control a data stream, including its file-position indicator, a pointer to its associated buffer (if any), an *error indicator* that records whether a read/write error occurred, and an *end-of-file* indicator that records whether the end of the file has been reached.

`fpos_t`

An object capable of recording all the information needed to uniquely specify every position within a file.

Macros

`NULL`

Expands to an implementation-defined null pointer constant.

`_IOFBF`
`_IOLBF`
`_IONBF`

Expand to integral constant expressions with distinct values, suitable for use as the third argument to the `setvbuf` function.

`BUFFSIZ`

Expands to an integral constant expression, which is the size of the buffer used by the `setbuf` function.

`EOF`

Expands to a negative integral constant expression that is returned by several functions to indicate end-of-file.

FOPEN_MAX

Expands to an integral constant expression that is the minimum number of files that the HP C compiler for your system guarantees can be open simultaneously.

FILENAME_MAX

Expands to an integral constant expression that is the size needed for an array of char large enough to hold the longest file name string that the HP C compiler for your system guarantees can be opened.

L_tmpnam

Expands to an integral constant expression that is the size needed for an array of char large enough to hold a temporary file name string generated by the tmpnam function.

SEEK_CUR
SEEK_END
SEEK_SET

Expand to integral constant expressions with distinct values; suitable for use as the third argument to the fseek function.

TMP_MAX

Expands to an integral constant expression that is the minimum number of unique file names that can be generated by the tmpnam function.

stderr
stdin
stdout

Expressions of type *pointer to* FILE that point to the FILE objects associated, respectively, with the standard error, input, and output streams.

File Operation Functions

int remove(const char *filename);

Makes the file whose name is pointed to by *filename* no longer accessible by that name. Any subsequent attempt to open that file using that name will fail. The remove function returns 0 if the operation succeeds, nonzero if it fails. If the file is open, the behavior of this function is implementation-defined.

```
int rename(const char *old, const char *new);
```

Renames the file from the name pointed to by *old* to the name pointed to by *new*. The file is no longer accessible by the old name. The `rename` function returns 0 if the operation succeeds, nonzero if it fails (in which case the file, if it existed, is still known by its original name). If the new file exists before `rename` is called, the behavior of this function is implementation-defined.

```
FILE *tmpfile(void);
```

Creates a temporary binary file that is automatically removed when it is closed or when program execution ends. If execution ends abnormally, whether an open temporary file is removed is implementation-dependent. The file is opened for update with `wb+` mode (see Table 9–1). The `tmpfile` function returns a pointer to the stream of the file that it created. If the file cannot be created, `tmpfile` returns a null pointer.

```
FILE *tmpnam(void);
```

Generates a valid file name that is different than the name of an existing file. Each call to `tmpnam`, up to `TMP_MAX` times, generates a different name. If `tmpnam` is called more than `TMP_MAX` times, the behavior is implementation-defined.

If the argument is a null pointer, the `tmpnam` function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to `tmpnam` can modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least `L_tmpnam chars`. The `tmpnam` function writes its result into that array and returns the argument as its value.

File Access Functions

```
int fclose(FILE *stream);
```

Flushes the stream pointed to by *stream* and closes the associated file. Any unwritten buffered data for the stream is delivered to the host environment to be written to the file. Any unread buffered data is discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated. The `fclose` function returns 0 if the stream was successfully closed, or it returns `EOF` if any errors are detected.

```
int fflush(FILE *stream);
```

If *stream* points to an output stream or an update stream in which the most recent operation was not input, the `fflush` function delivers any unwritten data to the host environment to be written to the file. Otherwise, the behavior is undefined. If *stream* is a null pointer, `fflush` flushes all output or update streams in which the most recent operation was not input. The `fflush` function returns 0 if the operation is successful, or it returns `EOF` if a write error occurs.

```
FILE *fopen(const char *filename, const char *mode);
```

Opens the file pointed to by *filename* and associates a stream with it. The argument *mode* points to a string beginning with one of the character sequences listed in Table 9–1. Additional characters can follow these sequences.

Table 9–1 File Modes

Mode	Description
`r`	open text file for reading
`w`	truncate to zero length or create text file for writing
`a`	append; open or create text file for writing at end-of-file
`rb`	open binary file for reading
`wb`	truncate to zero length or create binary file for writing
`ab`	append; open or create binary file for writing at end-of-file
`r+`	open text file for update (reading and writing)
`w+`	truncate to zero length or create text file for update
`a+`	append; open or create text file for update, writing at end-of-file
`r+b` *or* `rb+`	open binary file for update (reading and writing)
`w+b` *or* `wb+`	truncate to zero length or create binary file for update
`a+b` *or* `ab+`	append; open or create binary file for update, writing at end-of-file

The `fopen` function returns a pointer to the object controlling the stream. If the open operation fails, `fopen` returns a null pointer.

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

Opens the file pointed to by *filename* and associates the stream pointed to by *stream* with it. The *mode* argument is used in the same way as with the `fopen` function. The `freopen` function first tries to close any file associated

with the specified stream. Failure to close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared.

The primary use of `freopen` is to change the file associated with a standard text stream (`stderr`, `stdin`, or `stdout`), because those identifiers need not be modifiable lvalues to which the value returned by the `fopen` function can be assigned.

The `freopen` function returns a pointer to the object controlling the stream. If the open operation fails, `freopen` returns a null pointer.

```
void setbuf(FILE *stream, char *buf);
```

Except that it returns no value, the `setbuf` function is equivalent to the `setvbuf` function invoked with the values `_IOFBF` for *mode* and `BUFSIZ` for *size*, or (if *buf* is a null pointer) with the value `_IONBF` for *mode*.

```
int setvbuf(FILE *stream, char *buf, int mode size_t size);
```

Associates a buffer with an input or an output file. The `setvbuf` function can be used only after the stream pointed to by *stream* has been associated with an open file and before any other operation is performed on the stream. The argument *mode* determines how *stream* is to be buffered:

- `IOFBF` causes I/O to be fully buffered.
- `IOLBF` causes I/O to be line buffered.
- `IONBF` causes I/O to be unbuffered.

If *buf* is not a null pointer, the array it points to can be used instead of a buffer allocated by the `setvbuf` function. The size of the array is specified by *size*. The contents of the array at any time are indeterminate. The `setvbuf` function returns 0 if successful, or nonzero if an invalid value is specified for *mode* or if the request cannot be honored.

Formatted Input/Output Functions

```
int fprintf(FILE *stream, const char *format, ...);
```

Writes output to the stream pointed to by *stream*, under control of the string pointed to by *format*, which specifies how subsequent arguments are converted for output. If there are an insufficient number of arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored. The `fprintf` function returns when the end of the format string is encountered. The `fprintf` function returns the number of characters transmitted, or it returns a negative value if an output error occurred.

See your HP C library routine documentation for more information.

```
int fscanf(FILE *stream, const char *format, ...);
```

Reads input from the stream pointed to by *stream*, under control of the string pointed to by *format*, which specifies the allowable input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are an insufficient number of arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

The `fscanf` function returns the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, `fscanf` returns the number of input items assigned, which can be fewer than provided for, or even 0, if there is an early matching failure.

See your HP C library routine documentation for more information.

```
int printf(const char *format, ...);
```

Equivalent to the `fprintf` function except that `printf` writes formatted output to the standard output stream (`stdout`).

```
int scanf(const char *format, ...);
```

Equivalent to the `fscanf` function except that `scanf` reads formatted input from the standard input stream (`stdin`).

```
int sprintf(char *s, const char *format, ...);
```

Equivalent to the `fprintf` function except that the argument *s* specifies an array, rather than a stream, into which the generated output will be written. A null character is written at the end of the characters written. If copying takes place between objects that overlap, the behavior is undefined. The `sprintf` function returns the number of characters written into the array, not counting the terminating null character.

```
int sscanf(const char *s, const char *format, ...);
```

Equivalent to the `fscanf` function except that the argument *s* specifies a string, rather than a stream, from which the input will be read. Reaching the end of the string is equivalent to the `fscanf` function encountering end-of-file. If copying takes place between objects that overlap, the behavior is undefined.

```
#include <stdarg.h>
int vfprintf(FILE *stream, const char *format, va_list arg);
```

Equivalent to the `fprintf` function with the variable argument list replaced by *arg*, which must have been initialized by the `va_start` macro

(and possibly subsequent va_arg calls). The vfprintf function does not invoke the va_end macro.

```
#include <stdarg.h>
int vprintf(const char *format, va_list arg);
```

Equivalent to the printf function with the variable argument list replaced by *arg*, which must have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vprintf function does not invoke the va_end macro.

```
#include <stdarg.h>
int vsprintf(char *s, const char *format, va_list arg);
```

Equivalent to the sprintf function with the variable argument list replaced by *arg*, which must have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vsprintf function does not invoke the va_end macro.

Character Input/Output Functions

```
int fgetc(FILE *stream);
```

Returns the next character (if there is one) as an unsigned char converted to an int, from the input stream pointed to by *stream*, and advances the associated file-position indicator for the stream (if defined). If the stream is at end-of-file, the end-of-file indicator for the stream is set, and fgetc returns EOF. If a read error occurs, the error indicator is set, and fgetc returns EOF.

```
char *fgets(char *s, int n, FILE *stream);
```

Reads at most one less than the number of characters specified by *n* from the stream pointed to by *stream* into the array pointed to by *s*. No additional characters are read after a new-line character (which is retained) or after the end-of-file. A null character is written immediately after the last character read into the array.

The fgets function returns *s* if successful. If the end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

```
int fputc(int c, FILE *stream);
```

Writes the character *c* (converted to an unsigned char) to the output stream pointed to by *stream*, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator

appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream. The `fputc` function returns the character written. If a write error occurs, the error indicator for the stream is set, and `fputc` returns `EOF`.

```
int fputs(const char *s, FILE *stream);
```

Writes the string pointed to by *s* to the stream pointed to by *stream*. The terminating null character is not written.

The `fputs` function returns `EOF` if a write error occurs. Otherwise, it returns a nonnegative value.

```
int getc(FILE *stream);
```

Equivalent to the `fgetc` function, but if it is implemented as a macro it can evaluate *stream* more than once. For this reason, the argument should never be an expression with side effects.

```
int getchar(void);
```

Equivalent to the `getc` function with the argument `stdin`.

```
char *gets(char *s);
```

Reads characters from the input stream pointed to by `stdin` into the array pointed to by *s*, until the end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

The `fgets` function returns *s* if successful. If the end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

```
int putc(int c, FILE *stream);
```

Equivalent to the `fputc` function, but if it is implemented as a macro it can evaluate *stream* more than once. For this reason the argument should never be an expression with side effects.

```
int putchar(int c);
```

Equivalent to the `putc` function with the second argument `stdout`.

```
int puts(const char s);
```

Writes the string pointed to by *s* to the stream pointed to by stdout, and appends a new-line character to the output. The terminating null character is not written. The puts function returns EOF if a write error occurs. Otherwise, it returns a nonnegative value.

```
int ungetc(int c, FILE *stream);
```

Pushes a character *c* (converted to an unsigned char) back into the input stream pointed to by *stream*, and leaves the stream positioned before the character. The pushed back characters are returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call to a file positioning function for that stream (fseek, fsetpos, or rewind) discards any pushed-back characters.

One pushback is guaranteed, even if there has been no previous activity on the file. The ungetc function returns the converted pushed-back character, or it returns EOF if the operation fails.

Direct Input/Output Functions

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Reads into the array pointed to by *ptr* up to *nmemb* elements of size *size* from the stream pointed to by *stream*. The file-position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file-position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

The fread function returns the number of elements successfully read, which may be less than *nmemb* if a read error or end-of-file is encountered. If *size* or *nmemb* is 0, fread returns 0, and the contents of the array and the state of the stream are unchanged.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Writes from the array pointed to by *ptr* up to *nmemb* elements of size *size* to the stream pointed to by *stream*. The file-position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file-position indicator for the stream is indeterminate.

The fwrite function returns the number of elements successfully written, which is less than *nmemb* only if a write error is encountered.

File Positioning Functions

`int fgetpos(FILE *stream, fpos_t *pos);`

Stores the current value of the file-position indicator for the stream pointed to by *stream* into the object pointed to by *pos*. The value stored contains unspecified information used by the `fsetpos` function to return the stream to its position at the time of the call to `fgetpos`.

If successful, the `fgetpos` function returns 0. On failure, `fgetpos` returns nonzero and stores an implementation-defined positive value in `errno`.

`int fseek(FILE *stream, long int offset, int whence);`

Sets the file-position indicator to the specified byte offset in the stream pointed to by *stream*.

For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*, which is one of the following:

- The beginning of the file if *whence* is `SEEK_SET`
- The current value of the file-position indicator if *whence* is `SEEK_CUR`
- The end of the file if *whence* is `SEEK_END`

For a text stream, either *offset* is 0 or it is a value returned by an earlier call to the `ftell` function on the same stream and *whence* is `SEEK_SET`.

A successful call to `fseek` clears the end-of-file indicator for the stream and reverses any effects of the `ungetc` function on the same stream. After an `fseek` call, the next operation on an update stream can be either input or output. The `fseek` function returns nonzero only for a request that cannot be satisfied.

`int fsetpos(FILE *stream, const fpos_t *pos);`

Sets the file-position indicator for the stream pointed to by *stream* according to the value of the object pointed to by *pos*, which is a value obtained from an earlier call to the `fgetpos` function on the same stream.

A successful call to `fsetpos` clears the end-of-file indicator for the stream and reverses any effects of the `ungetc` function on the same stream. After an `fsetpos` call, the next operation on an update stream can be either input or output.

If successful, the `fsetpos` function returns 0. On failure, `fsetpos` returns nonzero and stores an implementation-defined positive value in `errno`.

```
long int ftell(FILE *stream);
```

Gets the current value of the file-position indicator for the stream pointed to by *stream*. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, its file-position indicator contains unspecified information used by the `fseek` function for returning the file-position indicator for the stream to its position at the time of the call to `ftell`. The difference between two such return values is not necessarily a meaningful measure of the number of characters written or read.

If successful, the `ftell` function returns the current value of the file-position indicator for the stream. On failure, `ftell` returns −1L and stores an implementation-defined positive value in `errno`.

```
void rewind(FILE *stream);
```

Sets the file-position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to the following, except that the error indicator for the stream is also cleared:

```
(void)fseek(stream, 0L, SEEK_SET)
```

The `rewind` function returns no value.

Error-Handling Functions

```
void clearerr(FILE *stream);
```

Clears the end-of-file and error indicators for the stream pointed to by *stream*. The `clearerr` function returns no value.

```
int feof(FILE *stream);
```

Tests the end-of-file indicator for the stream pointed to by *stream*. The `feof` function returns nonzero only if the end-of-file indicator is set for `stream`.

```
int ferror(FILE *stream);
```

Tests the error indicator for the stream pointed to by *stream*. The `ferror` function returns nonzero only if the end-of-file indicator is set for `stream`.

```
void perror(const char *s);
```

Maps the error number in the integer expression `errno` to an error message. It writes the following sequence of characters to the standard error stream:

1. The string pointed to by *s* followed by a colon (:) and a space (if *s* is not a null pointer and the character pointed to by *s* is not the null character)

2. An appropriate error message string followed by a new-line character

The contents of the error message strings are the same as those returned by the `strerror` function with argument `errno`, which are implementation-defined. The `perror` function returns no value.

9.14 General Utilities (<stdlib.h>)

The `<stdlib.h>` header file declares four types and several functions of general use, and defines several macros. The functions perform string conversion, random number generation, searching and sorting, memory management, and similar tasks.

Types

`size_t`

An unsigned integral type of the result of the `sizeof` operator.

`wchar_t`

An integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

`div_t`

A structure type that is the type of the value returned by the `div` function.

`ldiv_t`

A structure type that is the type of the value returned by the `ldiv` function.

Macros

`NULL`

Expands to an implementation-defined null pointer constant.

`EXIT_FAILURE/EXIT_SUCCESS`

Expand to integral expressions for use as the argument to the `exit` function to return unsuccessful or successful termination status, respectively, to the host environment. These macros are useful as return values from the main function as well.

`RAND_MAX`

Expands to an integral constant expression whose value is the maximum value returned by the `rand` function.

MB_CUR_MAX

Expands to a positive integer expression whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_TYPE), and whose value is never greater than MB_LEN_MAX.

String Conversion Functions

```
double atof(const char *nptr);
```

Converts the string pointed to by *nptr* to double representation and returns the converted value. Except for its behavior when an error occurs, this function is equivalent to:

```
strtod(nptr, (char **)NULL)
```

```
int atoi(const char *nptr);
```

Converts the string pointed to by *nptr* to int representation and returns the converted value. Except for its behavior when an error occurs, this function is equivalent to:

```
(int)strtol(nptr, (char **)NULL, 10)
```

```
long int atol(const char *nptr);
```

Converts the string pointed to by *nptr* to long int representation and returns the converted value. Except for its behavior when an error occurs, this function is equivalent to:

```
strtol(nptr, (char **)NULL, 10)
```

```
double strtod(const char *nptr, char **endptr);
```

Converts the string pointed to by *nptr* to double representation.

See your HP C library routine documentation for a detailed description of this function.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Converts the string pointed to by *nptr* to long int representation.

See your HP C library routine documentation for a detailed description of this function.

```
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

Converts the string pointed to by *nptr* to unsigned long int representation.

See your HP C library routine documentation for a detailed description of this function.

Pseudo-Random Sequence Generation Functions

`int rand(void);`

Returns a sequence of pseudo-random integers in the range 0 to `RAND_MAX`.

`void srand(unsigned int seed);`

Uses the argument as a seed for a new sequence of pseudo-random integers to be returned by subsequent calls to `rand`. If `srand` is then called with the same seed value, the sequence of pseudo-random integers is repeated. If `rand` is called before any calls to `srand` are made, the sequence generated is the same as when `srand` is first called with a seed value of 1. The `srand` function returns no value.

Memory Management Functions

`void *calloc(size_t nmemb, size_t size);`

Allocates an area in memory for an array of *nmemb* items, each with size *size*. The area is initialized to all bits 0. The `calloc` function returns either a null pointer if unable to allocate, or a pointer to the allocated area.

`void free(void *ptr);`

Deallocates the memory area pointed to by *ptr* that was allocated by a previous `calloc`, `malloc`, or `realloc`. If *ptr* is null, no action occurs. No value is returned.

`void *malloc(size_t size);`

Allocates a contiguous area in memory for an object of size *size*. The area is not initialized. This function returns a pointer to the allocated area, or it returns a null pointer if unable to allocate.

`void *realloc(void *ptr, size_t size);`

Changes the size of the area pointed to by *ptr* to the number of bytes specified by *size*. If *ptr* is null, the behavior of `realloc` is identical to `malloc`. The contents of the area are unchanged up to the lesser of the old and new sizes. This function returns either a null pointer if unable to resize, or a pointer to the possibly moved reallocated area.

Communication with the Environment

`void abort(void);`

Causes abnormal program termination to occur, unless the `SIGABRT` signal is being caught and the signal handler does not return. The `abort` function cannot return to its caller.

`int atexit(void (*func)(void));`

Registers the function pointed to by *func* to be called without arguments at normal program termination. Up to 32 functions can be registered. The `atexit` function returns 0 if the registration succeeds; otherwise, it returns nonzero.

`void exit(int status);`

Causes normal program termination to occur. If a program executes more than one call to `exit`, the behavior is undefined. Upon execution, the following occurs:

1. All functions registered by `atexit` are called in the reverse order of their registration.
2. All open output streams are flushed, all open streams are closed, and all files created by `tmpfile` are removed.
3. Control is returned to the host environment. The value of *status* corresponds to an `errno` value:
   - If the value *status* is 0 or `EXIT_SUCCESS`, a *successful termination* status is returned.
   - If the value *status* is `EXIT_FAILURE`, an *unsuccessful termination* status is returned.
   - Otherwise, an *unsuccessful termination* status is returned.

`char *getenv(const char *name);`

Searches an environment list provided by the host environment.

See your HP C library routine documentation for a detailed description of this function.

`int *system(const char *string);`

Passes the string pointed to by *string* to the host environment for execution by a command processor. A null pointer can be specified to inquire whether a command processor exists. If the argument is a null pointer, the `system` function returns nonzero if a command processor is available or 0 if one is not available. If the argument is not a null pointer, the return value is the status returned by the command processor or 0 if a command processor is not available.

See your HP C library routine documentation for a detailed description of this function.

Searching and Sorting Utilities

```
void *bsearch(const void *key, const void *base,
  size_t nmemb, size_t size, int (*compar)
  (const void *, const void *));
```

Searches an array of *nmemb* objects for an element that matches the object pointed to by *key*. The first element of the array is pointed to by *base*; the size of each element is specified by *size*.

You must first sort the array in ascending order according to the function pointed to by *compar*. The `bsearch` function calls the specified comparison function pointed to by *compar* with two arguments that point to the objects being compared (the *key* object and an array element). The comparison function returns:

- An integer less than 0, if the first argument is less than the second argument
- An integer greater than 0, if the first argument is greater than the second argument
- An integer equal to 0, if the first argument equals the second argument

The `bsearch` function returns a pointer to the matching element of the array, or a null pointer if no match is found.

```
void qsort(void *base, size_t nmemb,
size_t size, int (*compar) (const void *,
const void *));
```

Sorts an array of *nmemb* objects in place. The first element of the array is pointed to by *base*; the size of each element is specified by *size*.

The contents of the array are sorted in ascending order according to a comparison function pointed to by `compar`, which is called with two arguments that point to the objects being compared. The comparison function returns:

- An integer less than 0, if the first argument is less than the second argument
- An integer greater than 0, if the first argument is greater than the second argument
- An integer equal to 0, if the first argument equals the second argument

If two compared elements are equal, their order in the sorted array is unspecified.

The `qsort` function returns no value.

Integer Arithmetic Functions

`int abs(int j);`

Returns the absolute value of an integer *j*.

`div_t div(int numer, int denom);`

Computes the quotient and remainder of the division of *numer* by *denom*. The `div` function returns a structure of type `div_t` containing the quotient and remainder:

```
int quot;    /* quotient  */
int rem;     /* remainder */
```

`long int labs(long int j);`

Returns the absolute value of a long integer *j*.

`ldiv_t ldiv(long int numer, long int denom);`

Similar to the `div` function, except that the arguments and the members of the returned structure (which has type `ldiv_t`) all have type `long int`.

Multibyte Character Functions

`int mblen(const char *s, size_t n);`

If *s* is not a null pointer, `mblen` determines the number of bytes comprising the multibyte character pointed to by *s*. The `mblen` function is equivalent to the following, except that the shift state of the `mbtowc` is not affected:

```
mbtowc((wchar_t *)0, s, n);
```

If *s* is a null pointer, the `mblen` function returns a nonzero value if multibyte character encodings have state-dependent encodings, and 0 if they do not.

If *s* is not a null pointer, the `mblen` function returns one of the following values:

- 0, if *s* points to the null character
- The number of bytes that comprise the multibyte character, if the next *n* or fewer bytes form a valid multibyte character
- −1, if they do not form a valid multibyte character

`int mbtowc(wchar_t *pwc, const char *s, size_t n);`

If *s* is not a null pointer, `mbtowc` determines the number of bytes comprising the multibyte character pointed to by *s*. It then determines the code for the value of type `wchar_t` that corresponds to that multibyte character. (The value of the code corresponding to the null character is 0.) If the multibyte

character is valid and *pwc* is not a null pointer, mbtowc stores the code in the object pointed to by *pwc*. At most, *n* bytes of the array pointed to by *s* are examined.

If *s* is a null pointer, the mbtowc function returns a nonzero value if multibyte character encodings have state-dependent encodings, and 0 if they do not.

If *s* is not a null pointer, the mbtowc function returns one of the following values:

- 0, if *s* points to the null character
- The number of bytes that comprise the converted multibyte character, if the next *n* or fewer bytes form a valid multibyte character
- −1, if they do not form a valid multibyte character

```
int wctomb(char *s, wchar_t wchar);
```

Determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is *wchar*, including any change in shift state. This function then stores the multibyte character representation in the array object pointed to by *s*, if *s* is not a null pointer. At most, MB_CUR_MAX characters are stored. If the value of *wchar* is 0, the wctomb function is left in the initial shift state.

If *s* is a null pointer, the wctomb function returns a nonzero value if multibyte character encodings have state-dependent encodings, and 0 if they do not.

If *s* is not a null pointer, the wctomb function returns one of the following values:

- −1, if the value of *wchar* does not correspond to a valid multibyte character
- the number of bytes that comprise the multibyte character corresponding to the value of *wchar*

Multibyte String Functions

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Converts a sequence of multibyte characters that begin in the initial shift state from the array pointed to by *s* into a sequence of corresponding codes, and stores not more than *n* codes into the array pointed to by *pwcs*. A null character is converted to a code value of zero. No multibyte characters that follow a null character are examined or converted. Each multibyte character is converted as if by a call to mbtowc, except that the shift state of mbtowc is not affected.

If an invalid multibyte character is encountered, the mbstowcs function returns (size_t) - 1. Otherwise, it returns the number of array elements modified, not including a terminating zero code, if any.

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Converts a sequence of codes that correspond to multibyte characters from the array pointed to by *pwcs* into a sequence of multibyte characters that begins in the initial shift state, and stores these multibyte characters into the array pointed to by *s*. The conversion stops if a multibyte character would exceed the limit of *n* total bytes or if a null character is stored.

Each code is converted as if by a call to wctomb, except that the shift state of wctomb is not affected.

If a code is encountered that does not correspond to a valid multibyte character, the wcstombs function returns (size_t) - 1. Otherwise, it returns the number of bytes modified, not including a terminating null character, if any.

9.15 String Processing (<string.h>)

The <string.h> header file declares one type and several functions, and defines one macro useful for manipulating character arrays that other objects treat as character arrays.

There are two kinds of string functions declared. The first, with names beginning with str, manipulate character arrays; the second, with names beginning with mem, manipulate other objects treated as character arrays. Except for memmove, function behavior is undefined if copying takes place between overlapping objects.

Type

size_t

An unsigned integral type of the result of the sizeof operator.

Macro

NULL

Expands to an implementation-defined null pointer constant.

Functions

`void *memcpy(void *s1, const void *s2, size_t n);`

Copies *n* characters from the object pointed to by *s2* to the object pointed to by *s1*. The function returns *s1*.

`void *memmove(void *s1, const void *s2, size_t n);`

Copies *n* characters from the object pointed to by *s2* to the object pointed to by *s1*. Copying takes place as if the *n* characters from the object pointed to by *s2* are first copied into a temporary array of *n* characters that does not overlap the object pointed to by *s1* and *s2*, and then the *n* characters from the temporary array are copied into the object pointed to by *s1*. The `memmove` function returns *s1*.

`void *memchr(const void *s, int c, size_t n);`

Locates the first occurrence of *c* (converted to an unsigned `char`) in the first *n* unsigned characters of the object pointed to by *s*. The `memchr` function returns a pointer to the located character, or a null pointer if the character was not found.

`int memcmp(const void *s1, const void *s2, size_t n);`

Compares the first *n* characters of the object pointed to by *s1* to the first *n* characters of the object pointed to by *s2*. The `memcmp` function returns an integer less than, equal to, or greater than 0, depending on whether the object pointed to by *s1* is less than, equal to, or greater than the object pointed to by *s2*.

`void *memset(void *s, int c, size_t n);`

Copies the value of *c* (converted to an unsigned char) into each of the first *n* characters pointed to by *s*. The function returns *s*.

`char *strcpy(char *s1, const char *s2);`

Copies the string pointed to by *s2* (including the terminating null character) to the string pointed to by *s1*. The `strcpy` function returns *s1*.

`char *strncpy(char *s1, const char *s2, size_t n);`

Copies no more than *n* characters from the string pointed to by *s2* to the string pointed to by *s1*, up to but not including the null terminator of the string pointed to by *s2*; returns *s1*. If the string pointed to by *s2* is less than *n* characters, `strncpy` pads the copy with null characters.

```
char *strcat(char *s1, const char *s2);
```

Appends a copy of the the string pointed to by *s2* (including the terminating null character) to the end of the string pointed to by *s1*. The `strcat` function returns *s1*. The first character of *s2* overwrites the null character of *s1*.

```
char *strncat(char *s1, const char *s2, size_t n);
```

Appends no more than *n* characters from the string pointed to by *s2* (up to but not including a null character) to the string pointed to by *s1*. The `strncat` function returns *s1*. The first character of *s2* overwrites the null character of *s1*. A terminating null character is appended to the result. The first character of *s2* overwrites the null character of *s1*.

```
int strcmp(const char *s1, const char *s2);
```

Compares the string pointed to by *s1* to the string pointed to by *s2*. The `strcmp` function returns an integer less than, equal to, or greater than 0, depending on whether the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*.

```
int strcoll(const char *s1, const char *s2);
```

Compares the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the `LC_COLLATE` category of the current locale (see Section 9.6). The `strcoll` function returns an integer less than, equal to, or greater than 0, depending on whether the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*, when both are interpreted as appropriate to the current locale.

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Compares no more than *n* characters from the string pointed to by *s1* to the string pointed to by *s2*. The strings are compared until a null character is encountered, the strings differ, or *n* is reached. The `strncmp` function returns an integer less than, equal to, or greater than 0, depending on whether the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*.

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*.

See your HP C library routine documentation for a detailed description of this function.

```
char *strchr(const char *s, int c);
```

Locates the first occurrence of *c* (converted to a `char`) in the string pointed to by *s*. The terminating null character is considered to be part of the string. The function returns a pointer to the located character, or a null pointer if the character was not found.

```
size_t strcspn(const char *s1, const char *s2);
```

Computes the length of the maximum initial segment of the string pointed to by *s1* that consists entirely of characters not found in the string pointed to by *s2*. The `strcspn` function returns the length of the segment.

```
char *strpbrk(const char *s1, const char *s2);
```

Locates the first occurrence in the string pointed to by *s1* of any character from the string pointed to by *s2*. The function returns a pointer to the character, or a null pointer if no character in *s1* occurs in *s2*.

```
char *strrchr(const char *s, int c);
```

Locates the last occurrence of *c* (converted to a `char`) in the string pointed to by *s*. The terminating null character is considered to be part of the string. The function returns a pointer to the located character, or a null pointer if the character was not found.

```
size_t strspn(const char *s1, const char *s2);
```

Computes the length of the maximum initial segment of the string pointed to by *s1* that consists entirely of characters from the string pointed to by *s2*. The `strspn` function returns the length of the segment.

```
char *strstr(const char *s1, const char *s2);
```

Locates the first occurrence in the string pointed to by *s1* of the sequence of characters (excluding the terminal null character) in the string pointed to by *s2*. The `strstr` function returns a pointer to the located string, or a null pointer if the string was not found. If *s2* points to a string of zero length, the function returns *s1*.

```
char *strtok(const char *s1, char *s2);
```

Breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *s2*. The first call to `strtok( )` skips characters, looking for the first one that is not in *s2*. The function keeps track of its position in the string pointed to by *s1* between calls and, as successive calls are made, the function works through this string, identifying the text token following the one identified by the previous call. When the function finds a character in *s1* that matches a

character in *s2*, it replaces the character in *s1* with a null character. The `strtok` function returns a pointer to the first character of the token, or a null pointer if there is no token.

```
char *strerror(int errnum);
```

Maps the error number in `errnum` to an error message string; returns a pointer to the string. The string pointed to must not be modified by the program, but can be overwritten by a subsequent call to `strerror`.

```
size_t strlen(const char *s);
```

Computes the length of the string pointed to by *s*. The function returns the number of characters that precede the terminating null character.

9.16 Type-generic Math (<tgmath.h>)

The header `<tgmath.h>` includes the headers `<math.h>` and `<complex.h>` and defines several type-generic macros.

Of the `<math.h>` and `<complex.h>` functions without an f (`float`) or l (`long double`) suffix, several have one or more parameters whose corresponding real type is `double`. For each such function, except `modf`, there is a corresponding type-generic macro.[1] The parameters whose corresponding real type is `double` in the function synopsis are generic parameters. Use of the macro invokes a function whose corresponding real type and type domain are determined by the arguments for the generic parameters.[2]

Note

The type-generic implementation of the absolute value function (`fabs`) is not available for complex types in this release. You must use the type-specific names (`cabs`, `cabsf`, `cabsl`) instead.

[1] Like other function-like macros in Standard libraries, each type-generic macro can be suppressed to make available the corresponding ordinary function.

[2] If the type of the argument is not compatible with the type of the parameter for the selected function, the behavior is undefined.

9.16.1 Real-Type Determination

Use of the macro invokes a function whose generic parameters have the corresponding real type determined as follows:

1. First, if any argument for generic parameters has type `long double`, the type determined is `long double`.
2. Otherwise, if any argument for generic parameters has type `double` or is of integer type, the type determined is `double`.
3. Otherwise, the type determined is `float`.

9.16.2 Unsuffixed Functions in (<math.h>) and (<complex.h>) with the Same Name

For each unsuffixed function in `<math.h>` for which there is a function in `<complex.h>` with the same name except for a `c` prefix, the corresponding type-generic macro (for both functions) has the same name as the function in `<math.h>`. The corresponding type-generic macro for `fabs` and `cabs` is `fabs`. These functions are:

<math.h> function	<complex.h> function	type-generic macro
acos	cacos	acos
asin	casin	asin
atan	catan	atan
acosh	cacosh	acosh
asinh	casinh	asinh
atanh	catanh	atanh
cos	ccos	cos
sin	csin	sin
tan	ctan	tan
cosh	ccosh	cosh
sinh	csinh	sinh
tanh	ctanh	tanh
exp	cexp	exp
log	clog	log
pow	cpow	pow
sqrt	csqrt	sqrt
fabs	cabs	fabs

If at least one argument for a generic parameter is complex, then use of the macro invokes a complex function; otherwise, use of the macro invokes a real function.

9.16.3 Unsuffixed Functions in (<math.h>) with no c-prefixed Counterpart in (<complex.h>)

For each unsuffixed function in <math.h> without a c-prefixed counterpart in <complex.h>, the corresponding type-generic macro has the same name as the function. These type-generic macros are:

```
atan2         fma           llround       remainder
cbrt          fmax          log10         remquo
ceil          fmin          log1p         rint
copysign      fmod          log2          round
erf           frexp         logb          scalbn
erfc          hypot         lrint         scalbln
exp2          ilogb         lround        tgamma
expm1         ldexp         nearbyint     trunc
fdim          lgamma        nextafter
floor         llrint        nexttoward
```

If all arguments for generic parameters are real, then use of the macro invokes a real function; otherwise, use of the macro results in undefined behavior.

9.16.4 Unsuffixed Functions in (<complex.h>) that are not c-prefixed Counterparts to Functions in (<math.h>)

For each unsuffixed function in <complex.h> that is not a c-prefixed counterpart to a function in <math.h>, the corresponding type-generic macro has the same name as the function:

```
carg          conj          creal
cimag         cproj
```

Use of the macro with any real or complex argument invokes a complex function.

9.16.5 Example

Consider the following declarations:

```
#include <tgmath.h>
int n;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

Given these declarations, functions invoked by use of type-generic macros are as follows:

```
   macro use              invokes
---------------       ----------------------------
exp(n)                exp(n), the function
acosh(f)              acoshf(f)
sin(d)                sin(d), the function
atan(ld)              atanl(ld)
log(fc)               clogf(fc)
sqrt(dc)              csqrt(dc)
pow(ldc, f)           cpowl(ldc, f)
remainder(n, n)       remainder(n, n), the function
nextafter(d, f)       nextafter(d, f), the function
nexttoward(f, ld)     nexttowardf(f, ld)
copysign(n, ld)       copysignl(n, ld)
ceil(fc)              undefined behavior
rint(dc)              undefined behavior
fmax(ldc, ld)         undefined behavior
carg(n)               carg(n), the function
cproj(f)              cprojf(f)
creal(d)              creal(d), the function
cimag(ld)             cimagl(ld)
cabs(fc)              cabsf(fc)
carg(dc)              carg(dc), the function
cproj(ldc)            cprojl(ldc)
```

9.16.6 Imaginary Arguments

Type-generic macros that accept complex arguments also accept imaginary arguments. If an argument is imaginary, the macro expands to an expression whose type is real, imaginary, or complex, as appropriate for the particular function: if the argument is imaginary, then the types of `cos`, `cosh`, `fabs`, `carg`, `cimag`, and `creal` are real; the types of `sin`, `tan`, `sinh`, `tanh`, `asin`, `atan`, `asinh`, and `atanh` are imaginary; and the types of the others are complex.

Given an imaginary argument, each of the type-generic macros `cos`, `sin`, `tan`, `cosh`, `sinh`, `tanh`, `asin`, `atan`, `asinh`, `atanh` is specified by a formula in terms of real functions:

```
cos(iy)   = cosh(y)
sin(iy)   = i sinh(y)
tan(iy)   = i tanh(y)
cosh(iy)  = cos(y)
sinh(iy)  = i sin(y)
tanh(iy)  = i tan(y)
asin(iy)  = i asinh(y)
atan(iy)  = i atanh(y)
asinh(iy) = i asin(y)
atanh(iy) = i atan(y)
```

9.17 Date and Time (<time.h>)

The <time.h> header file defines two macros, and declares four types and several functions for manipulating time and date information. Some functions process local time, which may differ from calendar time because of time zone.

Types

size_t

An unsigned integral type of the result of the sizeof operator.

clock_t
time_t

Arithmetic types capable of representing times.

struct tm

Holds the components of a calendar time, called the *broken-down time*. The structure contains the following members:

```
int tm_sec;      /*  seconds after the minute -- [0,61]        */
int tm_min;      /*  minutes after the hour -- [0,59]          */
int tm_hour;     /*  hours since midnight -- [0,23]            */
int tm_mday;     /*  day of the month -- [1,31]                */
int tm_mon;      /*  months since January -- [0,11]            */
int tm_year;     /*  years since 1900                          */
int tm_wday;     /*  days since Sunday -- [0,6]                */
int tm_yday;     /*  days since January 1 -- [0,365]           */
int tm_isdst;    /*  Daylight Saving Time flag -- 0 if         */
                 /*  DST not in effect; positive if it is;     */
                 /*  negative if information is not available.  */
```

Macros

NULL

Expands to an implementation-defined null pointer constant.

CLOCKS_PER_SEC

The number per second of the value returned by the clock function.

Time Conversion Functions

char *asctime(const struct tm *timeptr);

Converts a broken-down time in the structure pointed to by *timeptr* into a 26-character string in the form of this example:

```
Sat Sep 08 08:10:32 1990\n\0
```

A pointer to the string is returned.

```
char *ctime(const time_t *timer);
```

Converts the calendar time pointed to by *timer* to local time in a string of the form generated by the `asctime` function. A pointer to the string is returned. The `ctime` function is equivalent to the following:

```
asctime(localtime(timer))
```

```
struct tm *gmtime(const time_t *timer);
```

Converts the calendar time pointed to by *timer* into a broken-down time expressed as Coordinated Universal Time (UTC). The `gmtime` function returns a pointer to the broken-down time, or a null pointer if UTC is not available.

```
struct tm *localtime(const time_t *timer);
```

Converts the calendar time pointed to by *timer* into a broken-down time expressed as local time. The `localtime` function returns a pointer to the broken-down time.

```
size_t strftime(char *s, size_t maxsize, const char *format, const
struct tm *timeptr);
```

Places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. The format string consists of zero or more conversion specifiers and ordinary multibyte characters. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. Each conversion specifier is replaced by the appropriate characters as shown in Table 9–2. The appropriate characters are determined by the `LC_TIME` category of the current locale and by the values contained in the structure pointed to by *timeptr*.

Table 9–2 strftime Conversion Specifiers

Specifier	Replaced by
%a	The locale's abbreviated weekday name
%A	The locale's full weekday name
%b	The locale's abbreviated month name
%B	The locale's full month name
%c	The locale's appropriate date and time representation
%d	The day of the month as a decimal number (01 – 31)

(continued on next page)

Table 9–2 (Cont.) strftime Conversion Specifiers

Specifier	Replaced by
%H	The hour (24-hour clock) as a decimal number (00 – 23)
%I	The hour (12-hour clock) as a decimal number (01 – 12)
%j	The day of the year as a decimal number (001 – 366)
%m	The month as a decimal number (01 – 12)
%M	The minute as a decimal number (00 – 59)
%p	The locale's equivalent of the AM/PM designations associated with a 12-hour clock
%S	The second as a decimal number (00 – 61)
%U	The week number of the year (the first Sunday as the first day of week 1) as a decimal number (00 – 53)
%w	The weekday as a decimal number (0 [Sunday] – 6)
%W	The week number of the year (the first Monday as the first day of week 1) as a decimal number (00 – 53)
%x	The locale's appropriate date representation
%X	The locale's appropriate time representation
%y	The year without century as a decimal number (00 – 99)
%Y	The year with century as a decimal number
%Z	The time zone name or abbreviation, or by no characters if no time zone can be determined
%%	%

If the total number of resulting characters including the terminating null character is not more than `maxsize`, the `strftime` function returns the number of characters placed into the array pointed to by *s*, not including the terminating null character. Otherwise, 0 is returned, and the array contents are indeterminate.

Time Manipulation Functions

```
clock_t clock(void);
```

Determines the processor time used. The `clock` function returns the processor time used by the program since the beginning of an event related to the program invocation. To determine the time in seconds, divide the return value by the value of the `CLOCKS_PER_SEC` macro. If the processor time is not available or cannot be represented, the value returned is `(clock_t)-1`. (To measure the time spent in a program, call the `clock`

function at the start of the program and subtract the return value from that of subsequent calls.)

`double difftime(time_t time1, time_t time0);`

Returns the difference between the two calendar times *time1* and *time0*, expressed in seconds, as a `double`.

`time_t mktime(struct tm *timeptr);`

Converts the broken-down time, expressed as local time, in the structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by the `time` function (that is, a value of type `time_t`), which it returns. If the calendar time cannot be represented, the value `(time_t)-1` is returned.

The original values of the `tm_wday` and `tm_yday` time components are ignored, and the original values of the other components are not restricted to the ranges indicated in the previous discussion of `struct_tm`. Upon successful completion of the function, the values of the `tm_wday` and `tm_yday` components are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated in the discussion of `struct_tm`. The final value of `tm_wday` is not set until `tm_mon` and `tm_year` are determined.

`time_t time(time_t *timer);`

Returns the current calendar time. If the calendar time is not available, the value `(time_t)-1` is returned.

A

Language Syntax Summary

This section summarizes the syntax of the C language, using the syntax of the ANSI C Standard. Syntactic categories are indicated with **bold** type, and literal words or characters are indicated with monospaced, nonitalicized type. A colon following a syntactic category introduces its definition. Alternative definitions are listed on separate lines, or are prefaced by the words ''one of.'' An optional element is indicated by the subscript $_{opt}$. For example, the following line indicates an optional expression enclosed in braces:

{ *expression*$_{opt}$ }

The section numbers shown in parentheses refer to the section of the *American National Standard for Information Systems-Programming Language C* (document number: X3.159-1989) that discusses that part of the language.

A.1 Lexical Grammar

A.1.1 Tokens

token: (§3.1)

keyword
identifier
constant
string-literal
operator
punctuator

preprocessing-token: (§3.1)

header-name
identifier
pp-number
character-constant
string-literal
operator

punctuator
each nonwhite-space character that cannot be one of the above

A.1.2 Keywords

keyword: (§3.1.1) one of

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
_Bool      _Complex
```

(Alpha, I64)

A.1.3 Identifiers

identifier: (§3.1.2)

identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-nondigit:

nondigit
other implementation-defined characters

nondigit: §3.1.2 one of

```
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: (§3.1.2) one of

```
0  1  2  3  4  5  6  7  8  9
```

A.1.4 Constants

constant: (§3.1.3)

floating-constant
integer-constant
enumeration-constant
character-constant

floating-constant: (§3.1.3.1)

decimal-floating-constant
hexadecimal-floating-constant

decimal-floating-constant

fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

hexadecimal-floating-constant

hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part floating-suffix$_{opt}$
hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part floating-suffix$_{opt}$

fractional-constant: (§3.1.3.1)

digit-sequence$_{opt}$. *digit-sequence*
digit-sequence .

exponent-part: (§3.1.3.1)

e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

sign: (§3.1.3.1) one of

+ -

digit-sequence: (§3.1.3.1)

digit
digit-sequence digit

hexadecimal-fractional-constant:

hexadecimal-digit-sequence$_{opt}$. *hexadecimal-digit-sequence*
hexadecimal-digit-sequence .

binary-exponent-part:

p *sign*$_{opt}$ *digit-sequence*

P *sign*$_{opt}$ *digit-sequence*

hexadecimal-digit-sequence:

hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit

floating-suffix: (§3.1.3.1) one of

```
f l F L
```

integer-constant: (§3.1.3.2)

decimal-constant integer-suffix$_{opt}$
octal-constant integer-suffix$_{opt}$
hexadecimal-constant integer-suffix$_{opt}$

decimal-constant: (§3.1.3.2)

nonzero-digit
decimal-constant digit

octal-constant: (§3.1.3.2)

0
octal-constant octal-digit

hexadecimal-constant: (§3.1.3.2)

0x *hexadecimal-digit*
0X *hexadecimal-digit*
hexadecimal-constant hexadecimal-digit

nonzero-digit: (§3.1.3.2) one of

```
1  2  3  4  5  6  7  8  9
```

octal-digit: (§3.1.3.2) one of

```
0  1  2  3  4  5  6  7
```

hexadecimal-digit: (§3.1.3.2) one of

```
0  1  2  3  4  5  6  7  8  9
a  b  c  d  e  f
A  B  C  D  E  F
```

integer-suffix: (§3.1.3.2)

unsigned-suffix long-suffix$_{opt}$

long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: (§3.1.3.2) one of

```
u U
```

long-suffix: (§3.1.3.2) one of

```
l L
```

enumeration-constant: (§3.1.3.3)

identifier

character-constant: (§3.1.3.4)

' *c-char-sequence*'
L' *c-char-sequence*'

c-char-sequence: (§3.1.3.4)

c-char
c-char-sequence c-char

c-char: (§3.1.3.4)

any member of the source character set except
the single-quote ('), backslash (\), or new-line character
escape-sequence

escape-sequence: (§3.1.3.4)

simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence: (§3.1.3.4) one of

```
\'  \"  \?  \\
\a  \b  \f  \n  \r  \t  \v
```

octal-escape-sequence: (§3.1.3.4)

\ *octal-digit*
\ *octal-digit octal-digit*
\ *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:(§3.1.3.4)

\x *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

A.1.5 String Literals

string-literal: (§3.1.4)

"*s-char-sequence*$_{opt}$"
L"*s-char-sequence*$_{opt}$"

s-char-sequence: (§3.1.4)

s-char
s-char-sequence s-char

s-char: (§3.1.4)

any member of the source character set except
the double-quote ("), backslash (\), or new-line character
escape-sequence

A.1.6 Operators

operator: (§3.1.5) one of

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

A.1.7 Punctuators

punctuator: (§3.1.6) one of

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

A.1.8 Header Names

header-name: (§3.1.7)

<h-char-sequence>
"*q-char-sequence*"

h-char-sequence: (§3.1.7)

h-char

h-char-sequence h-char

h-char: (§3.1.7)

any member of the source character set except
the new-line character and >

q-char-sequence: (§3.1.7)

q-char
q-char-sequence q-char

q-char: (§3.1.7)

any member of the source character set except
the new-line character and "

A.1.9 Preprocessing Numbers

pp-number: (§3.1.8)

digit
. digit
pp-number digit
pp-number nondigit
pp-number e *sign*
pp-number E *sign*
pp-number .

A.2 Phrase Structure Grammar

A.2.1 Expressions

primary-expression: (§3.3.1)

identifier
constant
string-literal
(expression)

postfix-expression: (§3.3.2)

primary-expression
postfix-expression [expression]
postfix-expression (*argument-expression-list*$_{opt}$)
postfix-expression . identifier

postfix-expression -> identifier
postfix-expression ++
postfix-expression - -

argument-expression-list: (§3.3.2)

assignment-expression
argument-expression-list , assignment-expression

unary-expression: (§3.3.3)

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

unary-operator: (§3.3.3) one of

```
&   *   +   –   ~   !
```

cast-expression: (§3.3.4)

unary-expression
(type-name) cast-expression

multiplicative-expression: (§3.3.5)

cast-expression
*multiplicative-expression * cast-expression*
multiplicative-expression / cast-expression
multiplicative-expression % cast-expression

additive-expression: (§3.3.6)

multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

shift-expression: (§3.3.7)

additive-expression
shift-expression << additive-expression
shift-expression >> additive-expression

relational-expression: (§3.3.8)

shift-expression
relational-expression < shift-expression
relational-expression > shift-expression

relational-expression <= shift-expression
relational-expression >= shift-expression

equality-expression: (§3.3.9)

relational-expression
equality-expression = = relational-expression
equality-expression != relational-expression

AND-expression: (§3.3.10)

equality-expression
AND-expression & equality-expression

exclusive-OR-expression: (§3.3.11)

AND-expression
exclusive-OR-expression ^ AND-expression

inclusive-OR-expression: (§3.3.12)

exclusive-OR-expression
inclusive-OR-expression | exclusive-OR-expression

logical-AND-expression: (§3.3.13)

inclusive-OR-expression
logical-AND-expression && inclusive-OR-expression

logical-OR-expression: (§3.3.14)

logical-AND-expression
logical-OR-expression || logical-AND-expression

conditional-expression: (§3.3.15)

logical-OR-expression
logical-OR-expression ? expression : conditional-expression

assignment-expression: (§3.3.16)

conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: (§3.3.16) one of

```
=   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
```

expression: (§3.3.17)

assignment-expression

expression , assignment-expression

constant-expression: (§3.4)

conditional-expression

A.2.2 Declarations

declaration: (§3.5)

declaration-specifiers init-declarator-list$_{opt}$;

declaration-specifiers: (§3.5)

storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$

init-declarator-list: (§3.5)

init-declarator
init-declarator-list , init-declarator

init-declarator: (§3.5)

declarator
declarator = initializer

storage-class-specifier: (§3.5.1)

typedef
extern
static
auto
register

type-specifier: (§3.5.2)

void
char
short
int
long
float
double
signed
unsigned _Bool _Complex *(Alpha, I64)*

struct-or-union-specifier

enum-specifier
typedef-name

struct-or-union-specifier: (§3.5.2.1)

struct-or-union identifier$_{opt}$ *{ struct-declaration-list }*
struct-or-union identifier

struct-or-union: (§3.5.2.1)

struct
union

struct-declaration-list: (§3.5.2.1)

struct-declaration
struct-declaration-list struct-declaration

struct-declaration: (§3.5.2.1)

specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list: (§3.5.2.1)

type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list: (§3.5.2.1)

struct-declarator
struct-declarator-list , struct-declarator

struct-declarator: (§3.5.2.1)

declarator
declarator$_{opt}$: *constant-expression*

enum-specifier: (§3.5.2.2)

enum *identifier*$_{opt}$ *{enumerator-list }*
enum *identifier*$_{opt}$ *{enumerator-list , }*
enum *identifier*

enumerator-list: (§3.5.2.2)

enumerator
enumerator-list , enumerator

enumerator: (§3.5.2.2)

enumeration-constant

enumeration-constant = constant-expression

type-qualifier: (§3.5.3)

const
volatile

declarator: (§3.5.4)

pointer$_{opt}$ *direct-declarator*

direct-declarator: (§3.5.4)

identifier
(declarator)
direct-declarator [*constant-expression*$_{opt}$]
direct-declarator (parameter-type-list)
direct-declarator (*identifier-list*$_{opt}$)

pointer: (§3.5.4)

* *type-qualifier-list*$_{opt}$
* *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list: (§3.5.4)

type-qualifier
type-qualifier-list type-qualifier

parameter-type-list: (§3.5.4)

parameter-list
parameter-list , ...

parameter-list: (§3.5.4)

parameter-declaration
parameter-list , parameter-declaration

parameter-declaration: (§3.5.4)

declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{opt}$

identifier-list: (§3.5.4)

identifier
identifier-list , identifier

type-name: (§3.5.5)

specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator: (§3.5.5)

pointer
pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator: (§3.5.5)

(abstract-declarator)
direct-abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
direct-abstract-declarator$_{opt}$ (*parameter-type-list*$_{opt}$)

typedef-name: (§3.5.6)

identifier

initializer: (§3.5.7)

assignment-expression
{ initializer-list }
{ initializer-list , }

initializer-list: (§3.5.7)

initializer
initializer-list , initializer

A.2.3 Statements

statement: (§3.6)

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

labeled-statement: (§3.6.1)

identifier : statement
case *constant-expression : statement*
default : *statement*

compound-statement: (§3.6.2)

{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list: (§3.6.2)

declaration

declaration-list declaration

statement-list: (§3.6.2)

statement
statement-list statement

expression-statement: (§3.6.3)

expression$_{opt}$;

selection-statement: (§3.6.4)

if *(expression) statement*
if *(expression) statement* else *statement*
switch *(expression) statement*

iteration-statement: (§3.6.5)

while *(expression) statement*
do *statement* while *(expression)* ;
for (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*

jump-statement: (§3.6.6)

goto *identifier* ;
continue ;
break ;
return *expression*$_{opt}$;

A.2.4 External Definitions

translation-unit: (§3.7)

external-declaration
translation-unit external-declaration

external-declaration: (§3.7)

function-definition
declaration

function-definition: (§3.7.1)

declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

A.3 Preprocessing Directives

preprocessing-file: (§3.8)

group$_{opt}$

group: (§3.8)

group-part
group group-part

group-part: (§3.8)

pp-tokens$_{opt}$ *new-line*
if-section
control-line

if-section: (§3.8.1)

if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group: (§3.8.1)

#if *constant-expression new-line group*$_{opt}$
#ifdef *identifier new-line group*$_{opt}$
#ifndef *identifier new-line group*$_{opt}$

elif-groups: (§3.8.1)

elif-group
elif-groups elif-group

elif-group: (§3.8.1)

#elif *constant-expression new-line group*$_{opt}$

else-group: (§3.8.1)

#else *new-line group*$_{opt}$

endif-line: (§3.8.1)

#endif *new-line*

control-line:

#include *pp-tokens new-line* (§3.8.2)
#define *identifier replacement-list new-line* (§3.8.3)
#define *identifier (identifier-list)*$_{opt}$ *replacement-list new-line* (§3.8.3)
#undef *identifier new-line* (§3.8.3)
#line *pp-tokens new-line* (§3.8.4)

#error *pp-tokens*$_{opt}$ *new-line* (§3.8.5)
#pragma *pp-tokens*$_{opt}$ *new-line* (§3.8.6)
new-line (§3.8.7)

lparen: (§3.8.3)

the left parenthesis character without preceding white space

replacement-list: (§3.8.3)

pp-tokens$_{opt}$

pp-tokens: (§3.8)

preprocessing-token
pp-tokens preprocessing-token

new-line: (§3.8)

the new-line character

B

ANSI Conformance Summary

HP C conforms to the ANSI standard for the Programming Language C, as specified by the X3J11 Technical Committee and documented in the *American National Standard for Information Systems–Programming Language C* (document number: X3.159-1989). HP C has successfully passed the Plum-Hall test suite for ANSI conformance. In strict ANSI C mode, the HP C compiler is a conforming implementation as described by the ANSI C Standard in Section 1.7, *Compliance*: '' A *conforming hosted implementation* shall accept any strictly conforming program. A conforming implementation can have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program. ''

The ANSI C Standard defines a strictly conforming program as:

'' A *strictly conforming program* shall use only those features of the language and library specified in this Standard. It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit. ''

'' An implementation shall be accompanied by a document that defines all implementation-defined characteristics and all extensions. ''

As with most language definitions, the ANSI C Standard does not encompass the entire definition of the C language available within an implementation. The C implementations currently supported by HP include a number of features that are not defined in the ANSI C Standard.

The rest of this section describes the compiler's functionality in a format mirroring the outline of the ANSI C Standard. The relevant ANSI C Standard section number is shown in parentheses following each heading. If a heading from the ANSI C Standard is missing from this description, HP C conforms to the Standard exactly, without extension or implementation-defined behavior.

The following sections document only the extensions and implementation-defined portions of the HP C language. Together with the ANSI C Standard, this section completely specifies the HP C implementation of the C language. The ANSI C Standard is referred to as ‘‘the Standard’’ throughout this appendix.

B.1 Diagnostics (§2.1.1.3)

A diagnostic message is produced for the first violation of a syntax rule or constraint specified in the Standard. Subsequent violations are reported if they are not hidden by previous violations.

B.2 Hosted Environment (§2.1.2.2)

The semantics of the arguments to `main()`, including `envp`, are determined by the programming environment. See your platform-specific HP C documentation for information on arguments to `main()`.

B.3 Multibyte Characters (§2.2.1.2)

The shift states used for the encoding of multibyte characters are dependent on translation tables available on the local system. A particular character set is supported by the language if the local system’s translation tables support it.

B.4 Escape Sequences (§2.2.2)

Elements within a character constant or string literal of the source character set are mapped directly into the elements of the execution character set. Escape sequences other than those defined by the Standard are diagnosed with a warning and the backslash is ignored, so that the character constant’s or string literal’s value is the same as if the backslash were not present.

B.5 Translation Limits (§2.2.4.1)

Translation limits vary across platforms because of differences in the underlying machine architecture and operating systems. Otherwise, HP C avoids imposing translation limits.

The following lists show the only limits imposed in HP C. Translation limits listed in the Standard, but not in the following list, are not imposed in HP C:

- 32,767 characters in an internal identifier or a macro name
- 32,767 characters in a logical or physical source line

- 32,767 bytes in the representation of a string literal (this limit does not apply to string literals formed as a result of concatenation)

On Tru64 UNIX systems:

- 1023 significant initial characters in an external identifier. A warning is issued if such an identifier is truncated.

On OpenVMS systems:

- 31 significant initial characters in an external identifier. A warning is issued if such an identifier is truncated.
- 253 actual arguments or formal parameters to a function.
- 1012 bytes in a function argument list.

B.6 Numerical Limits (§2.2.4.2)

HP C's numerical limits are defined in the `limits.h` and `float.h` header files. These header files contain the implementation-defined values so that the following descriptions hold:

- There are 8 bits in a character of the execution character set.
- The representation and set of values for the type `char` are the same as that of type `signed char`. This equivalence can be changed from `signed char` to `unsigned char` with a command-line option.
- On OpenVMS systems, the representation and set of values for the types `int` and `signed int` are the same as that for type `long` (32 bits).
- On OpenVMS systems, the representation and set of values for the type `unsigned int` are the same as that for type `unsigned long` (32 bits).
- On Tru64 UNIX systems, the `long int` and `unsigned long int` types are 64 bits, while `int` and `unsigned int` are 32 bits.
- The representation and set of values for the type `long double` are the same as that for type `double` (64 bits).

Any limits not found in the previous list are defined as shown in the Standard.

B.7 Keywords (§3.1.1)

The `__inline`, `__unaligned`, and `__restrict` keywords are supported on OpenVMS Alpha systems and Tru64 UNIX systems.

All VAX C keywords are supported in VAX C mode. They are:

- `_align`
- `globaldef`
- `globalref`
- `globalvalue`
- `noshare`
- `readonly`
- `variant_struct`
- `variant_union`

The following keywords are accepted on Tru64 UNIX systems, but result in a warning:

- `_align`
- `noshare`
- `readonly`

On Tru64 UNIX systems, `globaldef` and initialized `globalvalue` declarations are treated as external definitions. `globalref` and uninitialized `globalvalue` declarations are treated as if they were declared `extern`.

Note

The `MAIN_PROGRAM` option is also available with the VAX C compatibility option on OpenVMS systems.

B.8 Identifiers (§3.1.2)

An identifier can include the character dollar sign ($). (A warning is given for this in strict ANSI mode.)

On Tru64 UNIX systems, case distinctions are always significant in an identifier with external linkage.

On OpenVMS systems, all identifier names with external linkage are converted to uppercase by default, but this can be controlled with a command-line option.

B.9 Linkages of Identifiers (§3.1.2.2)

An error is reported if, within a translation unit, the same identifier appears with both internal and external linkage.

B.10 Types (§3.1.2.5)

The type `char` and the type `signed char` have the same representation and set of values. (If the unsigned compile-time option is specified, then the types `char` and `unsigned char` have the same representation and set of values.)

B.11 Integer Constants (§3.1.3.2)

The digits 8 and 9 are permitted as valid octal digits in common C and VAX C modes, but a warning message is issued.

B.12 Character Constants (§3.1.3.4)

A character constant containing more than one character or wide character is diagnosed with a warning under the error-checking compiler option and is stored as an integer value. A character constant with more than one character is represented with the last character in the low-order byte for compatibility with common C. Representation of an integer character constant containing an octal or hexadecimal escape sequence not in the basic execution character set is the value specified by the octal or hexadecimal number in the escape sequence. (Its value is interpreted as a signed or unsigned `char`, depending on whether the unsigned compile-time option is in effect.)

The type of a wide character constant, `wchar_t`, is `unsigned int`.

B.13 String Literals (§3.1.4)

The Standard states that identical string literals need not be distinct, and any attempt to modify a string literal is undefined. Therefore, it is an error to modify either a character-string literal or wide-string literal.

B.14 Operators—Compound Assignment (§3.1.5)

The old form of compound assignment operators (such as =+, =-, =*, =/, and =%) are not defined in the Standard.[1] Therefore, in expressions of the form *expression =unary_op expression*, where the *=unary_op* would previously have been interpreted as an assignment operator, the *=unary_op* is now interpreted as two tokens: the assignment operator and the *unary_op*.

A warning message is issued if the error-checking option is specified for =-, =*, =& and =+ (with no intervening white space) to remind you of this change in meaning. Without the error-checking option, no message is issued.

B.15 Characters and Integers—Value-Preserving Promotions (§3.2.1.1)

Two different approaches to the implementation of integer promotion rules have been taken by earlier versions of C. The first approach is called *unsigned preserving*, in which `unsigned char` and `unsigned short` widen to `unsigned int`. The second approach is called *value preserving*, in which `unsigned char` and `unsigned short` widen to `signed int` if the value can be represented; otherwise they widen to `unsigned int`. The Standard specifies that integer promotions are to be value-preserving. This approach is followed in all modes except common C and VAX C mode, and results in a quiet change to programs depending on unsigned-preserving arithmetic conversions.

To aid the programmer in locating arithmetic conversions that depend on unsigned-preserving rules, any integer promotions of `unsigned char` and `unsigned short` to `int` that could be affected by the value-preserving approach for integer promotions are flagged with the error-checking option.

B.16 Signed and Unsigned Integer Conversions (§3.2.1.2)

If the value of an integer demoted to a signed integer is too large to be represented, the result is truncated with excess high-order bits discarded. This is compatible with common C and VAX C.

Conversions between signed and unsigned integers of the same size involve no representation change.

[1] Early versions of C allowed compound assignment operators to be written in reverse form (=+, =-, = *) instead of the defined order (+=, -=, *=). This old form leads to syntactic ambiguities for the compound assignment operators whose second operator was also a valid unary operator.

B.17 Floating and Integral Conversions (§3.2.1.3)

When an integer is converted to a floating-point number that cannot be represented exactly, the result of the conversion is the nearest value that can be represented exactly. This result is the natural result of the conversion on the hardware, and can be higher or lower than the original value.

When a floating-point number is converted at compile time to an integer or another floating-point type, and the result cannot be represented, the compiler issues a diagnostic message.

When an integral number or double floating-point number is converted to a floating-point number that cannot exactly represent the original value, the result is rounded to the nearest value of type `float`. (For details, see the architecture manual for your platform; for example, the *MIPS R-Series Processor Architecture Manual* or the *VAX Architecture Manual.*)

When demoting a `double` value to `float`, if the value being converted is in the range of values that can be represented, but not represented exactly, the result is the nearest higher or lower value. HP C rounds the result to the nearest representable `float` value.

Similar rounding is performed for demotions from `long double` to `double` or `float`.

B.18 Pointer Conversions (§3.2.2.3)

Even if two types have the same representation (such as `int` and `long`), they are still different types. This means that a pointer to `int` cannot be assigned to a pointer to `long` without using a cast operation.

This rule is relaxed in the common C and VAX C modes. Pointer conversions do not involve a representation change, but, because of alignment restrictions on some machines, access through an unaligned pointer can result in much slower access time, a machine exception, or unpredictable results.

B.19 Structure and Union Members (§3.3.2.3)

The result of accessing a union member different than the member holding a value depends on the data types of the members and their alignment within the union.

B.20 The sizeof Operator (§3.3.3.4)

The type of the `sizeof` operator is `size_t`. HP C defines this type, which is the type of integer required to hold the maximum size of an array, in the `<stddef.h>` header as `unsigned int`.

B.21 Cast Operators (§3.3.4)

The Standard specifies that a pointer can be converted to an integral type, but the size of the integer required and the result are implementation-defined. A pointer occupies the same amount of storage as objects of type `int` or `long` (or their `unsigned` equivalents). Therefore, a pointer can be converted to any of these integer types and back again without changing its value. No scaling takes place, and the representation of the value does not change.

Converting between a pointer and a shorter integer type, such as `char`, is similar to the conversion between an object of `unsigned long` type and a shorted integer type. The high-order bits of the pointer are discarded. Converting between a shorter integer and a pointer is similar to the conversion between the shorter integer type and `unsigned long`. The high-order bits of the pointer are filled with copies of the sign bit if the shorter integer type was signed. Messages are issued for cast operations of these types under the error-checking compiler option.

B.22 Multiplicative Operators (§3.3.5)

The Standard does not provide portable semantics for the division and remainder operators. HP C follows these semantics:

- If either operand of the division operator (/) is negative, the result is truncated toward zero (the largest integer of lesser magnitude than the algebraic quotient)
- If either operand of the remainder operator (%) is negative, the sign of the result is the same as the sign of the first operand (for common C, MIPS C, and VAX C compatibility)

The compiler issues a warning in the following cases of undefined behavior detected at compile time:

- Integer overflow
- Division by zero
- Remainder by zero

B.23 Additive Operators (§3.3.6)

Pointers to members of the same array can be subtracted. The result is the number of elements between the two array members. The type of the result is `ptrdiff_t`. HP C defines this type as `int`.

B.24 Bitwise Shift Operators (§3.3.7)

The result of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has a signed type, the value of the result is the shifted value of `E1` with the vacated high-order bits filled with a copy of `E1`'s sign bit (arithmetic shift).

B.25 Storage-Class Specifiers (§3.5.1)

The `register` storage-class specifier suggests that access to the object be as fast as possible. Specifying `register` is intended to give a variable an increased probability of being stored in a register. However, compiler register allocation techniques make using the `register` keyword obsolete. That is, HP C accepts and ignores all `register` requests.

B.26 Type Specifiers (§3.5.2)

The combination `long float` is supported as a synonym for `double` for compatibility with common C and VAX C. This combination results in a warning if compiled with the default mode or the strict ANSI mode.

B.27 Structure and Union Specifiers (§3.5.2.1)

The high-order bit position of an `int` bit field is not treated as a sign bit, except in the VAX C compatibility mode. In other words, the type `int` designates the same type as `unsigned int` for all bit-field types. In VAX C mode, the type `int` designates the same type as `signed int` for all bit-field types.

B.28 Variant Structures and Unions

Variant structures and unions are VAX C extensions that allow nested structures and unions to be declared as members of the enclosing aggregate. This eliminates the need to specify an intermediate qualifier when referring to those members. These capabilities are only available in VAX C mode.

Your platform-specific HP C documentation contains details about these extensions.

B.29 Structure Alignment

The alignment and size of a structure is affected by the alignment requirements and sizes of the structure components for each platform. A structure can begin on any byte boundary and occupy any integral number of bytes. However, individual architectures or operating systems can specify particular default alignment and padding requirements, which can be overridden by pragmas and command-line options.

OpenVMS Alpha and Tru64 UNIX

On OpenVMS Alpha and Tru64 UNIX systems, nonbit-field structure members are, by default, aligned on natural boundaries.

The default alignment of a structure is the maximum alignment required by any member within the structure. The structure is padded to ensure that the size of a structure, in bytes, is a multiple of its alignment requirement to achieve the appropriate alignment when the structure or union is a member of an array.

The components of a structure are laid out in memory in the order they are declared. The first component has the same address as the entire structure. Padding is introduced between components to satisfy the alignment requirements of individual components.

A bit field can have any integral type. However, the compiler issues a warning with the error-checking option if the type is anything other than `int`, `unsigned int`, or `signed int`. The presence of bit fields causes the alignment of the whole structure or union to be at least the same as that of the bit field's base type.

Bit fields (including zero-length bit fields) not immediately declared following other bit fields have the alignment requirement imposed by their base type. Bit fields are allocated within the alignment unit (of the same size as the bit field's base type) from low-order to high-order.

With `#pragma member_alignment` in effect, if a bit field immediately follows another bit field, the bits are packed into adjacent space in the same unit, if sufficient space remains. Otherwise, padding is inserted at the end of the first bit field and the second bit field is put into the next unit.

With `#pragma nomember_alignment` in effect, bit fields are allowed to span storage unit boundaries. Alpha systems default to `member_alignment` while VAX systems default to `nomember_alignment`.

Bit fields of base type `char` cannot be larger than 8 bits. Bit fields of base type `short` cannot be larger than 16 bits.

OpenVMS VAX

OpenVMS VAX systems do not require that structures or structure members be aligned on any particular boundaries; nonbit-field structure members are byte-aligned by default.

The components of a structure are laid out in memory in the order they are declared. The first component has the same address as the entire structure. Each additional component follows its predecessor in the immediately following byte.

Natural alignment of structure members, can be obtained by using the following pragma:

```
pragma member_alignment
```

The *HP C User's Guide for OpenVMS Systems* has examples and diagrams of OpenVMS VAX structure alignment.

Bit fields can have any integral type. However, the compiler issues a warning if */STANDARD=ANSI89* is specified, and the type is other than `int`, `unsigned int`, or `signed int`. Bit fields are allocated within the unit from low order to high order. If a bit field immediately follows another bit field, the bits are packed into adjacent space, even if this overflows into another byte. However, if an unnamed bit field is specified to have length 0, filler is added so the bit field immediately following starts on the next byte boundary.

The *HP C User's Guide for OpenVMS Systems* has examples and diagrams of OpenVMS VAX bit-field alignment.

B.30 Enumeration Specifiers (§3.5.2.2)

The Standard specifies that each enumerated type be compatible with an implementation-defined integer type. In HP C, each enumerated type is compatible with the `signed int` type.

B.31 Type Qualifiers (§3.5.3)

The `volatile` storage class is specified for those variables that can be modified in ways unknown to the compiler. Thus, if an object is declared volatile, every reference to the object in the source code results in a reference to memory in the object code.

B.32 Declarators (§3.5.4)

There is no internal limit on the number of pointer, function or array declarators that can modify an arithmetic, structure, union, or incomplete type.

B.33 Initialization (§3.5.7)

C allows initializers to be optionally surrounded by braces ({ }) when they are not logically necessary. This has resulted in aggregate initializers with partially ignored braces that are parsed differently depending on the type of parser implemented (bottom-up or top-down). The Standard has specified the top-down parse originally specified in Kernighan and Ritchie's *The C Programming Language*. Programs depending on a bottom-up parse (common C parse) of partially braced initializers can yield unexpected results. Even though this construct is allowed, a warning message is given to inform the user of ignored braces when in common C mode or if using the check option.

B.34 The switch Statement (§3.6.4.2)

There is no limit on the number of `case` labels in a `switch` statement.

B.35 External Object Definitions (§3.7.2)

In common C mode, all `extern` objects have file scope.

B.36 Conditional Inclusion (§3.8.1)

Previous preprocessors have allowed extraneous text after a preprocessor directive. For example:

```
#endif system1
```

However, the Standard has stated that the only text allowed after a preprocessing directive is a comment. Therefore, the HP C compiler issues a warning message if this syntax rule is violated.

The numeric value for character constants within `#if` and `#elif` directives matches the value obtained when an identical character constant occurs in expressions that are not part of these directives.

B.37 Source File Inclusion (§3.8.2)

Source files can be included using either a quoted path name (`#include "stdio.h"`) or bracketed path names (`#include <stdio.h>`). OpenVMS systems also support a method of including modules from a text library. See your platform-specific HP C documentation for the search-path algorithm for including source files.

B.38 Macro Replacement—Predefined Macro Names (§3.8.3)

In addition to the predefined macro names defined in the Standard, the HP C compiler defines other preprocessor macros for various identification purposes. When the compiler is invoked, the appropriate identification macros are defined depending on the operating system, architecture, language, compiler mode, and other environment variables. You can reference these macros in `#ifdef` preprocessor directives to isolate code that applies to a particular environment.

Each HP C platform can have additional predefined macros. See your platform-specific HP C documentation for more information.

Table B–1 shows the predefined macro names for Tru64 UNIX.

Table B–1 Tru64 UNIX Predefined Macro Names

	Macro Name
Operating system name:	unix
	__unix__
	__osf
	SYSTYPE_BSD
	_SYSTYPE_BSD
Architecture name:	__alpha
Product name:	__DECC
	__DECC_VER
	LANGUAGE_C
	__LANGUAGE_C__

Table B–2 shows the predefined macro names for OpenVMS VAX and Alpha systems. All forms are defined unless strict ANSI mode is in effect, in which case only the new spellings are defined.

Table B–2 OpenVMS VAX, OpenVMS Alpha, and I64 Predefined Macro Names

	New Spelling	Traditional Spelling
Operating system name:	__vms	vms
	__VMS	VMS
	__vms_version	vms_version
	__VMS_VERSION	VMS_ VERSION
Architecture name:	__vax *(VAX)*	vax *(VAX)*
	__VAX *(VAX)*	VAX *(VAX)*
	__alpha *(Alpha)*	—
	__ALPHA *(Alpha)*	—
	__Alpha_AXP *(Alpha)*	—
	__IA64 *(I64)*	—
	__IA64__ *(I64)*	—
	__32BITS *(Alpha)*	—
Product name:	__vaxc	vaxc
	__VAXC	VAXC
	__vax11c	vax11c
	__VAX11C	VAX11C
	__STDC__	—
	__DECC	—
	__DECC_VER	—
	__VMS_V6_RTL_COMPAT	—
Compiler Mode:	__DECC_MODE_STRICT	—
	__DECC_MODE_RELAXED	—
	__DECC_MODE_VAXC	—
	__DECC_MODE_COMMON	—
Floating-Point:	__D_FLOAT	—
	__G_FLOAT	—
	__IEEE_FLOAT *(Alpha)*	—

(continued on next page)

Table B–2 (Cont.) OpenVMS VAX, OpenVMS Alpha, and I64 Predefined Macro Names

	New Spelling	Traditional Spelling
	__X_FLOAT *(Alpha)*	—
Other:	__HIDE_FORBIDDEN_NAMES	—
	__INITIAL_POINTER_SIZE *(Alpha)*	—

You can explicitly define the macros in Table B–3 to control which C library routines are declared in header files and to obtain standards conformance checking. To define these macros use one of the following:

- -D flag *(Tru64 UNIX)*
- /DEFINE qualifier *(OpenVMS)*
- `#define` preprocessor directive

Table B–3 Library Routine Standards Conformance Macros—All platforms

Macro	Standard
_XOPEN_SOURCE_EXTENDED	XPG4-UNIX
_XOPEN_SOURCE	XPG4
_POSIX_C_SOURCE	POSIX
_ANSI_C_SOURCE	ISO C and ANSI C
_AES_SOURCE *(Tru64 UNIX)*	Application Environment Services
_OSF_SOURCE *(Tru64 UNIX)*	OSF compatibility
_VMS_V6_SOURCE *(OpenVMS)*	OpenVMS Version 6 compatibility
_DECC_V4_SOURCE *(OpenVMS)*	DEC C Version 4 compatibility

B.39 The ## Operator (§3.8.3.3)

The ## operator within a macro replacement list causes the two tokens on either side of the operator to be concatenated into a single token.

In common C and VAX C compatibility mode, comments can also concatenate two tokens because in these modes a comment is replaced by a null string after macro invocations.

This behavior is not supported in strict ANSI or default mode, where comments are replaced with a single space.

B.40 Error Directive (§3.8.5)

The `#error` directive causes an error message to be issued and the compilation to cease.

B.41 Pragma Directive (§3.8.6)

The Standard's approved method of adding extensions to the language is through the addition of pragmas. All unrecognized pragmas are diagnosed with an informational message. Supported pragmas vary across platforms. See your platform-specific HP C documentation for more information.

When only preprocessing a file, all pragmas recognized by HP C are written unaltered to the output.

B.42 Function Inline Expansion

Function inline expansion eliminates procedure-call overhead and allows general optimization methods to apply across the expanded code. Function inlining has advantages over macros in that arguments are evaluated only once, parentheses need not be overused to avoid problems with precedence, and the actual expansion can be controlled from the command line.

The following pragmas are provided to control function inline expansion:

```
#pragma inline (function_name [,function_name....])
#pragma noinline (function_name [,function_name....])
```

If a function is named in an `inline` directive, calls to it are expanded as inline code, if the function has the following properties:

- If a function is named in a `noinline` directive, calls to it are not expanded as inline code.
- If a function is not named in an `inline` or a `noinline` directive, the compiler uses a heuristic to perform inline expansion of calls where appropriate.
- The compiler issues an error if a function is named in both an `inline` and a `noinline` directive.

  If the noinline compiler option is used, it overrides all `inline` pragma directives.

Inline functions have the following properties:

- An inline function can be recursive, but only one level of inline expansion is performed if it is.
- Only calls from the source file containing the definition of the inlined function are expanded inline.
- The address of an inline function can be taken and expressions that imply the conversion of the inlined function name to an address are allowed.
- The use of the `varargs` package (allowing a function to take a variable number of arguments) is not allowed for inline functions.
- An inline function cannot be declared with an ellipsis in its argument list.

B.43 Linkage Pragmas

HP C supports the `#pragma linkage` and `#pragma use_linkage` preprocessor directives on OpenVMS Alpha systems.

These pragmas are used for defining special linkage characteristics and to associate these linkage characteristics with functions. See your platform-specific HP C documentation for more information.

B.44 Other Pragmas

The following pragmas are provided for VAX C compatibility mode only:

```
#pragma dictionary CDD_path
#pragma module title ident
```

These pragmas correspond to the `#dictionary` and `#module` directives, respectively.

See your platform-specific HP C documentation for additional pragmas supported on your system.

C

ASCII Equivalence Table

Figure C–1 shows the ASCII character set. Each character's octal, decimal, and hexadecimal value is shown.

Figure C–1 ASCII Equivalence Chart

Character	Oct	Dec	Hex
\0	00	0	0x0
\001	01	1	0x1
\002	02	2	0x2
\003	03	3	0x3
\004	04	4	0x4
\005	05	5	0x5
\006	06	6	0x6
\007	07	7	0x7
\b	010	8	0x8
\t	011	9	0x9
\n	012	10	0xA
\v	013	11	0xB
\f	014	12	0xC
\r	015	13	0xD
\016	016	14	0xE
\017	017	15	0xF
\020	020	16	0x10
\021	021	17	0x11
\022	022	18	0x12
\023	023	19	0x13
\024	024	20	0x14
\025	025	21	0x15
\026	026	22	0x16
\027	027	23	0x17
\030	030	24	0x18
\031	031	25	0x19
\032	032	26	0x1A
\033	033	27	0x1B
\034	034	28	0x1C
\035	035	29	0x1D
\036	036	30	0x1E
\037	037	31	0x1F
(space)	040	32	0x20
!	041	33	0x21
"	042	34	0x22
#	043	35	0x23
$	044	36	0x24
%	045	37	0x25
&	046	38	0x26
\	047	39	0x27
(	050	40	0x28
)	051	41	0x29
*	052	42	0x2A
+	053	43	0x2B
,	054	44	0x2C
_	055	45	0x2D
.	056	46	0x2E
/	057	47	0x2F
0	060	48	0x30
1	061	49	0x31
2	062	50	0x32
3	063	51	0x33
4	064	52	0x34
5	065	53	0x35
6	066	54	0x36
7	067	55	0x37
8	070	56	0x38
9	071	57	0x39
|	072	58	0x3A
;	073	59	0x3B
<	074	60	0x3C
=	075	61	0x3D
>	076	62	0x3E
?	077	63	0x3F

Character	Oct	Dec	Hex
—	0100	64	0x40
A	0101	65	0x41
B	0102	66	0x42
C	0103	67	0x43
D	0104	68	0x44
E	0105	69	0x45
F	0106	70	0x46
G	0107	71	0x47
H	0110	72	0x48
I	0111	73	0x49
J	0112	74	0x4A
K	0113	75	0x4B
L	0114	76	0x4C
M	0115	77	0x4D
N	0116	78	0x4E
O	0117	79	0x4F
P	0120	80	0x50
Q	0121	81	0x51
R	0122	82	0x52
S	0123	83	0x53
T	0124	84	0x54
U	0125	85	0x55
V	0126	86	0x56
W	0127	87	0x57
X	0130	88	0x58
Y	0131	89	0x59
Z	0132	90	0x5A
[	0133	91	0x5B
\	0134	92	0x5C
]	0135	93	0x5D
^	0136	94	0x5E
_	0137	95	0x5F
`	0140	96	0x60
a	0141	97	0x61
b	0142	98	0x62
c	0143	99	0x63
d	0144	100	0x64
e	0145	101	0x65
f	0146	102	0x66
g	0147	103	0x67
h	0150	104	0x68
i	0151	105	0x69
j	0152	106	0x6A
k	0153	107	0x6B
l	0154	108	0x6C
m	0155	109	0x6D
n	0156	110	0x6E
o	0157	111	0x6F
p	0160	112	0x70
q	0161	113	0x71
r	0162	114	0x72
s	0163	115	0x73
t	0164	116	0x74
u	0165	117	0x75
v	0166	118	0x76
w	0167	119	0x77
x	0170	120	0x78
y	0171	121	0x79
z	0172	122	0x7A
{	0173	123	0x7B
|	0174	124	0x7C
}	0175	125	0x7D
~	0176	126	0x7E
\177	0177	127	0x7F

ZK–8422A–GE

D

Common C Extensions Supported by HP C

HP C supports several common C (old-style C) extensions to ANSI-standard C. These extensions are recognized only when the common C compatibility option is used on the compiler command line. The common C extensions allow you to use the c89 compiler to compile code originally written for the portable C compiler (pcc).

The following sections describe the common C extensions available with the common C compatibility option. Extensions to the ANSI-standard C language are divided into two categories:

- Extensions compatible with ANSI C programs that produce diagnostic messages when compiled without the common C compatibility option
- Extensions incompatible with ANSI C programs, which could produce different compiler behavior when used without the common C compatibility option

D.1 Extensions Compatible with ANSI C

- Relaxed pointer and pointer/integer compatibility is allowed. That is, all pointer and integer types are compatible, and pointer types are compatible with each other regardless of the type of object they point to. Therefore, under the common C option, a pointer to `float` is compatible with a pointer to `int`.
- The digits 8 and 9 are valid in octal integer constants. (A warning message is issued by the compiler, however.)
- Bit-field data types may include `enum`, `short`, `char`, and `long`. The ANSI C Standard allows only `int`, `unsigned int`, or `signed int`.
- `long float` is recognized as a synonym for `double`.

- A third argument to the function `main()`, namely `char *envp[]`, is allowed.[1]

  When HP C is run in common C compatibility mode, the `main` function can accept a third parameter, the environment array *envp*. This array contains process information such as the user name and controlling information, and has no bearing on passing command-line arguments. Its primary use is during `exec` and `getenv` library function calls.

  See your platform-specific HP C documentation for more information about invoking the `main` function within your host environment.
- Text is allowed following the preprocessing directives `#else` and `#endif`.
- Address constants may be cast to `int`.
- Tentative definitions that exist at the completion of a compilation remain tentative to the linker, in accordance with the traditional model of definition resolution.
- Casts that do not cause a change in representation are legal as lvalues.
- Implicit function declarations are created at file level, rather than at block level.
- The types `int` and `long` are compatible.
- Taking the address of a variable with the `register` storage class is allowed.
- Block-level declarations of functions with `static` storage class are allowed.
- In array types, the element types are allowed to be incomplete.
- The type of a tentatively-defined variable is allowed to be incomplete at the end of the compilation unit. A warning is issued for this case.
- Values in case labels are allowed to have a pointer type.
- Trailing (extra) commas are allowed in enumeration lists.
- The semicolon following the last structure or union member may be omitted.
- Carriage returns are accepted and treated as white space.

[1] Parameters to the function `main()` are only checked in strict ANSI mode.

D.2 Extensions Incompatible with ANSI C

- Unsigned preserving rules apply. (`unsigned char` and `unsigned short` promote to `unsigned int`.)
- Comments are converted to no spaces instead of a single space to allow token concatenation. (The compiler attempts to concatenate the two adjacent tokens.)
- All `extern` objects have file scope.
- Macro parameters are recognized and replaced within string or character constants in the macro definition.
- During macro replacement, an argument's preprocessing tokens are not macro replaced before the macro is expanded.

  If the name of a macro being replaced is found during the rescan of the replacement list, it is macro replaced.
- Support for predefined macro names that do not conform to the ANSI C Standard (that is, that do not start with two underscores or an underscore followed by a capital letter).
- A preprocessor directive is only recognized as such if the beginning `#` character occurs in the first column of a line. Any preprocessor directives preceded by white space are ignored.
- `#ifdef` is treated as "`#if defined`"
- `#ifndef` is treated as "`#if !defined`"
- Comments in macro replacement lists behave like `##` operators when a valid token results after concatenation, except that adjoining white space is not deleted. If the resulting token is not valid, the comment in a macro replacement is deleted.
- Trigraphs are not recognized and replaced.

E

VAX C Extensions Supported by HP C

HP C supports several VAX C extensions to ANSI-standard C. These extensions are recognized only when the VAX C compatibility option is used on the compiler command line. The VAX C extensions allow you to use the HP C compiler to compile code originally written for the VAX C compiler.

The following sections describe the VAX C extensions available with the VAX C compatibility option. Extensions to the ANSI-standard C language are divided into two categories:

- Extensions compatible with ANSI C programs that produce diagnostic messages when compiled without the VAX C compatibility option
- Extensions incompatible with ANSI C programs, which could produce different compiler behavior when used without the VAX C compatibility option

E.1 Extensions Compatible with ANSI C

- VAX C specific pragmas are recognized.
- Relaxed pointer and pointer/integer compatibility is allowed. That is, all pointer and integer types are compatible, and pointer types are compatible with each other regardless of the type of object they point to. Therefore, under the VAX C option, a pointer to `float` is compatible with a pointer to `int`.
- The `#module` directive is allowed. (On Tru64 UNIX systems this directive produces a warning message and is ignored.)
- The `#dictionary` directive is allowed. (On Tru64 UNIX systems this directive produces a warning message and is ignored.)
- The module form of `#include` is allowed. (On Tru64 UNIX systems the module form of this directive produces an error.)

- Specifying `int` for the type of a bit field is equivalent to specifying `signed int` in VAX C mode.
- Built-in functions are recognized.
- The `main_program` option may be used to identify a particular function as the main function for a given program.

  When compiling in VAX C mode, another way to specify the main function in a program is to include the following option in the function definition:

  ```
  main_program
  ```

  This option is not a keyword, and it can be written uppercase or lowercase. The `main_program` option is useful for allowing a name other than `main` for the main program.

  In a prototype-style function definition, include `main_program` between the function declaration part and the left brace, as in the following example:

  ```
  char lower(int c_up)
  main_program
  {
     .
     .
     .
  }
  ```

  In an old-style function definition, include `main_program` in the same place as in the prototype style, but before any parameter declarations, as in the following example:

  ```
  char lower(c_up)
  main_program
  int c_up;
  {
     .
     .
     .
  }
  ```

  Both examples establish the function `lower` as the main function; execution begins there, regardless of the order in which the functions are linked.
- Bit-field data types may include `enum`, `short`, `char`, and `long`. The ANSI C standard allows only `int`, `unsigned int`, or `signed int`.
- The last member of a structure may be an array with no size specified.
- Two `struct` types or two `union` types are considered the same type if their sizes are the same.

- Block-level declarations of functions with `static` storage class are allowed.
- The address of a constant may be passed to a function.
- Taking the address of a variable with `register` storage class is allowed.
- A third argument to the function `main()`, namely `char *envp[]`, is allowed.

  When HP C is run in VAX C compatibility mode, the `main` function can accept a third parameter, the environment array *envp*. This array contains process information such as the user name and controlling information, and has no bearing on passing command-line arguments. Its primary use is during `exec` and `getenv` library function calls. [1]

  See your platform-specific HP C documentation for more information about invoking the `main` function within your host environment.
- `long float` is recognized as a synonym for `double`.
- Character constants containing multiple characters are packed in little endian order. For example, 'AB' is treated as 'B' << 8 + 'A' instead of 'A' << 8 + 'B'.
- Trailing (extra) commas are allowed in enumeration lists.
- The element type of an array may be incomplete.
- Carriage returns are accepted and treated as white space.

E.2 Extensions Incompatible with ANSI C

- Unsigned preserving rules apply. (`unsigned char` and `unsigned short` promote to `unsigned int`.)
- VAX C-specific predefined macros are recognized.
- VAX C-specific keywords are recognized.
- Macro parameters are recognized and are replaced as string or character constants in the macro definition.
- Comments are converted to no spaces instead of a single space to allow token concatenation. (The compiler attempts to concatenate the two adjacent tokens.)
- Comments in macro replacement lists behave like `##` operators when a valid token results after concatenation, except that adjoining white space is not deleted. If the resulting token is not valid, the comment in a macro replacement is deleted.

[1] Parameters to the function `main()` are only checked in strict ANSI mode.

- Trigraphs are not recognized or replaced.
- Variant structures and unions are allowed.

  Variant structure and union declarations allow reference to members of nested aggregates without having to refer to intermediate structure or union identifiers. When a variant structure or union declaration is nested within another structure or union declaration, the enclosed variant aggregate ceases to exist as a separate aggregate, and HP C copies its members to the enclosing aggregate.

  Variant structures and unions are declared using the `variant_struct` and `variant_union` keywords. The format of these declarations is the same as that for regular structures or unions, with the following exceptions:

  - Variant aggregates must be nested within other valid structure or union declarations.
  - A tag cannot be used in a variant aggregate declaration.
  - At least one member must be declared in the variant aggregate declaration, and it may not be declared as a pointer or an array.

  Initialization of a variant structure or union is the same as that of a normal structure or union.

  With the VAX C compatibility option, two structures or unions in an assignment operation need only have the same size, rather than requiring the same members and member types.

  The following example shows the format of a variable structure declaration, and how to reference members of a variant structure:

```
#include <stdio.h>
enum packet_type {TEXT, INTEGER};

/* This structure can contain either a text_packet  or an integer value.
   It can only contain one of these at a time, since they share the same
   storage. */
```

```
struct packet
{
    enum packet_type type;
    variant_union
    {
        variant_struct
        {
            int str_size;
            char *text;
        } text_packet;
        variant_struct
        {
          int value;
        } value_packet;

     } text_or_int;
} packet = {TEXT, 24 ,"I love the color purple"};

main()
{

    if (packet.type == TEXT)
        printf(" %s. \n",packet.text);
    else
        printf(" %d \n", packet.value);

    packet.type = INTEGER;
    packet.value = 42;

    printf(" The meaning of life, the universe, and everything is: %d. \n",
packet.value);

}
```

F

Universal Character Names for Identifiers

The following table lists the hexadecimal code values that are valid in universal character names in identifiers. Ranges that are part of the basic character sets are omitted.

Latin:	00AA, 00BA, 00C0-00D6, 00D8-00F6, 00F8-01F5, 01FA-0217, 0250-02A8, 1E00-1E9B, 1EA0-1EF9, 207F
Greek:	0386, 0388-038A, 038C, 038E-03A1, 03A3-03CE, 03D0-03D6, 03DA, 03DC, 03DE, 03E0, 03E2-03F3, 1F00-1F15, 1F18-1F1D, 1F20-1F45, 1F48-1F4D, 1F50-1F57, 1F59, 1F5B, 1F5D, 1F5F-1F7D, 1F80-1FB4, 1FB6-1FBC, 1FC2-1FC4, 1FC6-1FCC, 1FD0-1FD3, 1FD6-1FDB, 1FE0-1FEC, 1FF2-1FF4, 1FF6-1FFC
Cyrillic:	0401-040C, 040E-044F, 0451-045C, 045E-0481, 0490-04C4, 04C7-04C8, 04CB-04CC, 04D0-04EB, 04EE-04F5, 04F8-04F9
Armenian:	0531-0556, 0561-0587
Hebrew:	05B0-05B9, 05BB-05BD, 05BF, 05C1-05C2, 05D0-05EA, 05F0-05F2
Arabic:	0621-063A, 0640-0652, 0670-06B7, 06BA-06BE, 06C0-06CE, 06D0-06DC, 06E5-06E8, 06EA-06ED
Devanagari:	0901-0903, 0905-0939, 093E-094D, 0950-0952, 0958-0963
Bengali:	0981-0983, 0985-098C, 098F-0990, 0993-09A8, 09AA-09B0, 09B2, 09B6-09B9, 09BE-09C4, 09C7-09C8, 09CB-09CD, 09DC-09DD, 09DF-09E3, 09F0-09F1
Gurmukhi:	0A02, 0A05-0A0A, 0A0F-0A10, 0A13-0A28, 0A2A-0A30, 0A32-0A33, 0A35-0A36, 0A38-0A39, 0A3E-0A42, 0A47-0A48, 0A4B-0A4D, 0A59-0A5C, 0A5E, 0A74

```
Gujarati:       0A81-0A83,    0A85-0A8B,   0A8D,   0A8F-0A91,
                0A93-0AA8, 0AAA-0AB0,  0AB2-0AB3,  0AB5-0AB9,
                0ABD-0AC5, 0AC7-0AC9, 0ACB-0ACD, 0AD0, 0AE0

Oriya:          0B01-0B03,  0B05-0B0C,  0B0F-0B10, 0B13-0B28,
                0B2A-0B30, 0B32-0B33,  0B36-0B39,  0B3E-0B43,
                0B47-0B48, 0B4B-0B4D, 0B5C-0B5D, 0B5F-0B61

Tamil:          0B82-0B83,  0B85-0B8A,  0B8E-0B90, 0B92-0B95,
                0B99-0B9A,   0B9C,   0B9E-0B9F,    0BA3-0BA4,
                0BA8-0BAA,  0BAE-0BB5,  0BB7-0BB9, 0BBE-0BC2,
                0BC6-0BC8, 0BCA-0BCD

Telugu:         0C01-0C03, 0C05-0C0C,  0C0E-0C10,  0C12-0C28,
                0C2A-0C33,  0C35-0C39,  0C3E-0C44, 0C46-0C48,
                0C4A-0C4D, 0C60-0C61

Kannada:        0C82-0C83, 0C85-0C8C,  0C8E-0C90,  0C92-0CA8,
                0CAA-0CB3,  0CB5-0CB9,  0CBE-0CC4, 0CC6-0CC8,
                0CCA-0CCD, 0CDE, 0CE0-0CE1

Malayalam:      0D02-0D03, 0D05-0D0C,  0D0E-0D10,  0D12-0D28,
                0D2A-0D39,  0D3E-0D43,  0D46-0D48, 0D4A-0D4D,
                0D60-0D61

Thai:           0E01-0E3A, 0E40-0E5B

Lao:            0E81-0E82,  0E84,  0E87-0E88,   0E8A,   0E8D,
                0E94-0E97,  0E99-0E9F, 0EA1-0EA3, 0EA5, 0EA7,
                0EAA-0EAB, 0EAD-0EAE,  0EB0-0EB9,  0EBB-0EBD,
                0EC0-0EC4, 0EC6, 0EC8-0ECD, 0EDC-0EDD


Tibetan:        0F00, 0F18-0F19, 0F35, 0F37, 0F39, 0F3E-0F47,
                0F49-0F69, 0F71-0F84,  0F86-0F8B,  0F90-0F95,
                0F97, 0F99-0FAD, 0FB1-0FB7, 0FB9

Georgian:       10A0-10C5, 10D0-10F6

Hiragana:       3041-3093, 309B-309C

Katakana:       30A1-30F6, 30FB-30FC

Bopomofo:       3105-312C

CJK Unified Ideographs: 4E00-9FA5

Hangul:         AC00-D7A3

Digits:         0660-0669,  06F0-06F9,  0966-096F, 09E6-09EF,
                0A66-0A6F, 0AE6-0AEF,  0B66-0B6F,  0BE7-0BEF,
                0C66-0C6F,  0CE6-0CEF,  0D66-0D6F, 0E50-0E59,
                0ED0-0ED9, 0F20-0F33
```

Special characters:	00B5, 00B7, 02B0-02B8, 02BB, 02BD-02C1, 02D0-02D1, 02E0-02E4, 037A, 0559, 093D, 0B3D, 1FBE, 203F-2040, 2102, 2107, 210A-2113, 2115, 2118-211D, 2124, 2126, 2128, 212A-2131, 2133-2138, 2160-2182, 3005-3007, 3021-3029

Index

A

abort library function, 9–37
abs library function, 9–40
Abstract declarator
  cast, 6–16
  defined, 2–28
  example, 2–28
acos library function, 9–17
Active position, 1–26
+ (addition operator), 6–21
Additive inverse, 6–13
Additive operator, 6–21
  HP C behavior, B–9
Address-of operator (&), 6–15
& (address operator), 6–15
Aggregate array, 4–15
  See also Bracket operator ([ ])
Aggregate type, 3–11
Aggregate variant, E–4
Aliasing, 3–24
Alignment
  predefined constants, 2–22
Alignment unit, 3–15
__align modifier, 2–21
Allocation
  storage, 4–4
AND bitwise operator (&), 6–24
ANSI compatible extensions of HP C
  common C extensions, D–1
  VAX C extensions, E–1
ANSI conformance, B–1
ANSI C Standard
  document information, xiii
ANSI C standard limits, 1–24
argc
  main function argument, 5–14
Argument
  array as argument, 5–12
  command-line, 5–14
  Conversion, 5–12
  conversion of function, 6–37
  function as argument, 5–12
  function prototype, 5–8
  in #define preprocessor macros, 8–4
  passing by value, 5–11
  rules governing, 5–11
  to a function
    conversion, 5–12
  variable
    header file, 9–21
Argument promotions
  default, 6–10
argv
  main function argument, 5–14
Arithmetic
  complex, 9–2
  negation operator, 6–13
Arithmetic conversion
  usual, 6–33
Arithmetic type, 3–3
Array
  as argument, 5–12
  as expression, 6–7
  declaration of, 4–15
  initialization, 4–18, 4–33
  references, 6–7

Array (cont'd)
size determination, 3–13
variable length, 4–22
Array declaration
syntax, 4–15
Array pointer, 4–21
Array storage
row-major order, 4–18
Array subscripts, 3–13
Array type
discussed, 3–12
Arrow operator (->), 6–10
ASCII character set, 1–3
ASCII equivalence table, C–1
asctime library function, 9–50
asin library function, 9–17
<assert.h> header, 9–2
assert macro, 9–2
Assignment operator, 6–26 to 6–27
precedence of, 6–5
= (assignment operator), 6–26
+= (assignment operator), 6–26
-= (assignment operator), 6–26
*= (assignment operator), 6–26
Associativity of operator, 6–6
Asterisk operator (*), 4–13
atan2 library function, 9–17
atan library function, 9–17
atexit library function, 9–38
atof library function, 9–36
atoi library function, 9–36
atol library function, 9–36
auto class
defined, 2–14
example, 2–14
auto keyword
used in declaration inside block, 7–2
Automatic storage duration, 2–14

B

Backslash continuation character
in #define, 8–3
Backslash-newline continuation, 1–5
Basic concepts of C, 2–1
Basic data types, 3–2
Binary operator
additive, 6–21
bitwise, 6–24
defined, 1–11
equality, 6–23
logical, 6–24
multiplicative, 6–20
precedence of, 6–5
relational, 6–23
shift, 6–22
Bit field
creation of, 4–29
declaration, 4–28
declaration syntax, 4–28
OpenVMS VAX alignment, B–11
restrictions, 4–29
ULTRIX RISC alignment, B–10
Bit-field
common C data types, D–1
& (bitwise AND operator), 6–24
Bitwise negation operator (~), 6–15
Bitwise operator, 6–24
| (bitwise OR operator), 6–24
Bitwise shift operator, B–9
^ (bitwise XOR operator), 6–24
Block
defined, 2–2
example, 2–2
Block scope
defined, 2–4
example, 2–4
Block statement, 7–2
_Bool data type, 3–8
header file, 9–22
Boolean type and values
header file, 9–22
bool macro, 9–23
__bool_true_false_are_defined macro, 9–23
Bracket operator ([ ]), 6–7

[ ] (bracket operator), 6–7
break statement
  defined, 7–12
  from switch statement, 7–5
bsearch library function, 9–39
BUFFSIZ macro, 9–24

C

cabs library functions, 9–6
cacosh library functions, 9–4
cacos library functions, 9–3
calloc library function, 9–37
carg library functions, 9–7
case label, 7–5
casinh library functions, 9–5
casin library functions, 9–3
Cast operator, 6–15
  HP C behavior, B–8
( ) (cast operator), 6–15
catanh library functions, 9–5
catan library functions, 9–4
ccosh library functions, 9–5
ccos library functions, 9–4
ceil library function, 9–19
cexp library functions, 9–6
Character
  data type
    object, 4–10
  string, 4–15
    See also Array
Character constant
  defined, 1–19
  HP C specific, B–5
  value, 1–19
Character display
  defined, 1–26
Character escape sequence
  list, 1–21 to 1–22
Character object declaration, 4–10
Character processing
  header file, 9–8
Character set
  defined, 1–3
Character string, 3–9
Character type, 3–8
cimag library functions, 9–7
C language
  list of operators, 6–3
clearerr library function, 9–34
C lexicon
  grammar, 1–1
C library
  prototype, 5–10
clock library function, 9–52
CLOCKS_PER_SEC macro, 9–50
clock_t type, 9–50
clog library functions, 9–6
Command-line argument, 5–14
Comma operator (,), 6–27
  precedence of, 6–5
, (comma operator), 6–27
Comment, 1–8 to 1–9
Common C extensions of HP C, D–1
Common definition
  header file, 9–23
Compatible type
  categories, 2–9
  defined, 2–9
Compilation unit
  data sharing, 2–3
  defined, 1–2
  discussed, 2–3
Complete type, 2–8
<complex.h> header, 9–2
Complex arithmetic
  header file, 9–2
Complex data type, 3–10
_Complex data type, 3–10
complex macro, 9–3
_Complex_I macro, 9–3
Composite type
  conditions, 2–11
  defined, 2–11
Compound assignment operator, B–6
Compound literal, 6–29
Compound statement, 7–2

Conditional compilation, 8–8 to 8–10
Conditional inclusion, B–12
Conditional operator (?:)
defined, 6–25
precedence of, 6–5
?: (conditional operator), 6–25
Conditional statement, 7–4 to 7–8
Conforming implementation, B–1
Conforming program, B–1
conj library functions, 9–8
const
in pointer declaration, 4–13
in variable declaration, 4–13
Constant
character, 1–19
defined, 1–14
enumeration, 1–23
floating-point, 1–17
hexadecimal, 1–18
integer, 1–14
Constant expression
address constant, 6–29
arithmetic, 6–29
defined, 6–28
integral, 6–28
Constants
identifiers in #define macros, 8–4
Constructor expression, 6–29
const type qualifier
discussed, 3–20 to 3–21
example, 3–20
rules, 3–21
Continuation
logical line, 1–4
string, 1–5
string termination, 1–13
Continuation character
in #define, 8–3
continue statement, 7–11
Control characters, 1–4
Conventions
notational, xvii
Conversion, 6–32
arithmetic data-type, 6–32
function argument, 6–8
Conversion (cont'd)
function prototype present, 6–37
of data types, 6–32
usual arithmetic, 6–33
with cast operator, 6–15
cosh library function, 9–17
cos library function, 9–17
cpow library functions, 9–6
cproj library functions, 9–8
creal library functions, 9–8
csinh library functions, 9–5
csin library functions, 9–4
csqrt library functions, 9–7
ctanh library functions, 9–6
ctan library functions, 9–4
ctime library function, 9–51
<ctype.h> header, 9–8
Current object, 4–33

D

Data type
basic, 3–2
_Bool, 3–8
character, 3–8
complex, 3–10
conversion, 6–32
floating-point, 3–9
function, 5–2
function prototype, 5–8
integral, 3–6
introduction, 3–1
list, 3–2
range, 3–4
size, 3–4
__DATE__ predefined macro, 8–16
Date and time
header file, 9–50
Declaration
aggregate array, 4–15
C library prototype, 5–10
enumeration, 4–11
example, 4–3
format, 4–9
function, 5–6
function prototype, 5–8

Declaration (cont'd)
general syntax, 4–2
inside block, 7–2
structure, 4–24
syntax rules, 4–3
tentative tag declaration, 4–26
type definitions, 4–37
union, 4–24
Declarator, B–12
Declaring floating-point objects, 4–11
Decrement operator
-- postfix, 6–11
prefix, 6–13
-- prefix, 6–13
Default argument promotions, 6–10
default label, 7–5
Default widening convention, 5–12
#define directive, 8–2
defined operator, 8–11
Definition
function, 5–3
object, 4–4
Dereferencing pointer, 6–15
Derived type list, 3–11
Designation, 4–33
in initializer list, 4–33
Designator, 4–33
Diagnostic, B–2
Diagnostics
header file, 9–2
#dictionary directive, E–1
Difference between structure and union, 4–27
difftime library function, 9–53
Digraph sequence, 1–6
Directives
See Preprocessor directives
Division operator (/), 6–20
/ (division operator), 6–20
div library function, 9–40
div_t type, 9–35
do statement, 7–9
Dot operator (.), 6–10

E

EDOM macro, 9–10
#elif preprocessing directive, 8–8, 8–10
Ellipsis
in prototype, 5–9
else clause, 7–4
#else preprocessing directive, 8–8
common C extension, D–2
#endif preprocessing directive, 8–8
common C extension, D–2
Enumerated data type
declaration, 4–11
Enumerated type
discussed, 3–18
example, 3–18
Enumeration constant
defined, 1–23
syntax, 4–11
type, 3–18
Enumeration specifier, B–11
enum keyword, 4–11
envp
main function argument
common C extension, D–2
VAX C extension, E–3
EOF macro, 9–24
Equality operator, 6–23
= = (equality operator), 6–23
Equal-to operator (=), 6–23
ERANGE macro, 9–10
<errno.h> header, 9–10
errno macro, 9–10
Error Code
header file, 9–10
Error directive, B–16
#error preprocessing directive, 8–16
Errors
types, 1–2
Escape sequence, B–2
Evaluation order
of argument list, 5–11

Execution character set
defined, 1–3
list, 1–4
exit library function, 9–38
EXIT_FAILURE macro, 9–35
EXIT_SUCCESS macro, 9–35
exp library function, 9–18
Expression
assignment, 6–26
as statement, 7–3
binary
additive, 6–21
bitwise, 6–24
equality, 6–23
logical, 6–24
multiplicative, 6–20
relational, 6–23
shift, 6–22
comma, 6–27
compound literal, 6–29
conditional, 6–25
constant, 6–28
Constructor, 6–29
evaluation order, 2–7
postfix
array reference, 6–7
decrement operator, 6–11
function call, 6–8
increment operator, 6–11
structure reference, 6–10
syntax, 6–7
union reference, 6–10
primary
constant, 6–2
defined, 6–2
identifier, 6–2
parentheses, 6–2
syntax, 6–2
relational, 6–23
unary
addressed, 6–15
arithmetic negation, 6–13
bitwise negation, 6–15
cast, 6–15
increment and decrement, 6–13
Expression
unary (cont'd)
logical negation, 6–13
_Pragma, 6–19
sizeof, 6–17
__typeof__, 6–18
External declaration, 4–7
scope, 4–8
External definition, 4–7
External linkage, 2–11
External object definition, B–12
extern class, 2–15

F

fabs library function, 9–19
false macro
stdbool.h, 9–23
fclose library function, 9–26
feof library function, 9–34
ferror library function, 9–34
fflush library function, 9–27
fgetc library function, 9–30
fgetpos library function, 9–33
fgets library function, 9–30
File
header, 9–1
__FILE__predefined macro, 8–16
FILENAME_MAX macro, 9–25
File scope
defined, 2–4
example, 2–4
FILE type, 9–24
Floating point
conversion to integer, B–7
Floating-point
data type
declaration, 4–10
precision, 4–10
Floating-point constant
defined, 1–17
notation, 1–19
type, 1–18

Floating-Point constant
hexadecimal, 1–18
Floating-point object declaration, 4–11
Floating point type, 3–9
Floating point types
list, 3–9
float keyword, 4–10
floor library function, 9–19
fmod library function, 9–19
fopen library function, 9–27
FOPEN_MAX macro, 9–25
__forceinline modifier, 2–21
for statement, 7–9
Forward reference
defined, 2–23
example, 2–23
fpos_t type, 9–24
fprintf library function, 9–28
fputc library function, 9–30
fputs library function, 9–31
fread library function, 9–32
free library function, 9–37
freopen library function, 9–27
frexp library function, 9–18
fscanf library function, 9–29
fseek library function, 9–33
fsetpos library function, 9–33
ftell library function, 9–34
__func__ predeclared identifier, 8–18
Function
address, 5–12
address passing, 6–9
as argument, 5–12
call
defined, 5–1
syntax, 6–8
within macros, 8–6
C library
prototype, 5–10
declaration, 5–6
definition
argument, 5–11
argument conversion, 6–8
defined, 5–3
parameter, 5–11
Function (cont'd)
definition of
main_program option, E–2
identifier, 6–9
implicit declaration of, 5–8
main, 5–14
prototype, 5–8
syntax, 5–9
type, 5–2
undeclared, 6–8
variable parameter list, 5–9
Function argument
conversion, 6–37
to main, 5–14
Function definition
syntax, 5–3
Function inline expansion, B–16
Function prototype
defined, 5–8
Function prototypes
scope rules, 5–10
type conversion, 5–10
widening rules, 5–10
Function prototype scope
defined, 2–5
example, 2–5
Function scope
defined, 2–5
example, 2–5
Function type
discussed, 3–11
example, 3–11
fwrite library function, 9–32

G

getchar library function, 9–31
getc library function, 9–31
getenv library function, 9–38
gets library function, 9–31
gmtime library function, 9–51
goto statement, 7–11
Greater-than operator (>), 6–23

> (greater-than operator), 6–23
Greater-than or equal-to operator(>=), 6–23
>= (greater-than or equal-to operator), 6–23

H

Header file
  <assert.h>, 9–2
  complex.h>, 9–2
  <ctype.h>, 9–8
  defined, 1–24, 9–1
  <errno.h>, 9–10
  <limits.h>, 9–11
  <locale.h>, 9–11
  <math.h>, 9–16
  <setjmp.h>, 9–19
  <signal.h>, 9–20
  <stdarg.h>, 9–21
  stdbool.h>, 9–22
  <stddef.h>, 9–23
  <stdio.h>, 9–24
  <stdlib.h>, 9–35
  <string.h>, 9–42
  <time.h>, 9–50
Hexadecimal constant, 1–14
Hexadecimal floating-point constant, 1–18
Hosted Environment, B–2
HUGE_VAL macro, 9–16

I

Identifier
  defined, 1–6
  HP C, B–5
  linkage, B–5
    significant characters, 1–7
  predeclared
    __func__, 8–18
  rules, 1–7
#ifdef preprocessing directive, 8–8
if false, definition, 7–4
#ifndef preprocessing directive, 8–8
#if preprocessing directive, 8–8
  using the defined operator, 8–11
if statement, 7–4
if true, definition, 7–4
_Imaginary data type, 3–11
imaginary macro, 9–3
_Imaginary_I macro, 9–3
#include directive
  module form, E–1
#include preprocessing directive, 8–11
Including file
  C library prototype, 5–10
Including files, 8–11
Incompatible type, 2–10
Incomplete array declaration
  example, 3–13
  uses, 4–16
Incomplete type
  defined, 2–8
  example, 2–8
  forming with tags, 2–24
Increment operator
  ++ postfix, 6–11
  prefix, 6–13
  ++ prefix, 6–13
Indirection operator (*), 6–15
* (indirection operator), 6–15
!= (inequality operator), 6–23
INFINITY macro, 9–16
Initialization, 4–4
  array, 4–16, 4–18, 4–33
  C99 Standard, 4–33
  constraints, 4–5
  general, 4–5
  HP C behavior, B–12
  implicit, 4–5
  structure, 4–30, 4–33
  union, 4–32
Initializers
  syntax, 4–5
Initializing a pointer, 4–15
inline modifier, 2–17
__inline modifier, 2–17
Input/Output
  header file, 9–24

Integer
conversion to floating point, B–7
Integer constant
defined, 1–14
octal, B–5
rules, 1–14
Integer data type
declaration, 4–10
Integer object declaration, 4–10
Integer promotion, 6–33
unsigned preserving, B–6
value preserving, B–6
Integral type, 3–6
discussed, 3–6 to 3–7
Internal declaration, 4–7
Internal linkage, 2–11
Inverse
additive, 6–13
_IOFBF macro, 9–24
_IOLBF macro, 9–24
_IONBF macro, 9–24
isalnum library function, 9–9
isalpha library function, 9–9
iscntrl library function, 9–9
isdigit library function, 9–9
isgraph library function, 9–9
islower library function, 9–9
ISO C Standard, xiii
isprint library function, 9–9
ispunct library function, 9–9
isspace library function, 9–9
isupper library function, 9–10
isxdigit library function, 9–10
Iteration statement, 7–8 to 7–10

J

jmp_buf type, 9–19
Jump, nonlocal
header file, 9–19
Jump statement, 7–11 to 7–12

K

Keyword
break statement, 7–12
case label, 7–5
continue statement, 7–11
default label, 7–5
defined, 1–9
do statement, 7–9
else clause, 7–4
enum, 4–11
for statement, 7–9
goto statement, 7–11
if statement, 7–4
list, 1–9
return statement, 7–12
sizeof, 6–17
switch statement, 7–5
uses, 1–10
VAX C, B–4
void, 4–14
while statement, 7–8
Keywords
__align, 2–21
__forceinline, 2–21
inline, 2–17
__inline, 2–17

L

Label statement
case, 7–5
defined, 7–1
labs library function, 9–40
LC_ALL macro, 9–11
LC_COLLATE macro, 9–11
LC_MONETARY macro, 9–11
LC_NUMERIC macro, 9–11
LC_TIME macro, 9–11
LC_TYPE macro, 9–11
ldexp library function, 9–18
ldiv library function, 9–40

ldiv_t type, 9–35
<< (left shift operator), 6–22
Less-than operator (<), 6–23
< (less-than operator), 6–23
Less-than or equal-to operator (<=), 6–23
<= (less-than or equal-to operator), 6–23
Lexical error, 1–2
Lexicon of the language, 1–1
Library function, 9–1 to 9–53
abort, 9–37
abs, 9–40
acos, 9–17
asctime, 9–50
asin, 9–17
atan, 9–17
atan2, 9–17
atexit, 9–38
atof, 9–36
atoi, 9–36
atol, 9–36
bsearch, 9–39
cabs, 9–6
cacos, 9–3
cacosh, 9–4
calloc, 9–37
carg, 9–7
casin, 9–3
casinh, 9–5
catan, 9–4
catanh, 9–5
ccos, 9–4
ccosh, 9–5
ceil, 9–19
cexp, 9–6
cimag, 9–7
clearerr, 9–34
clock, 9–52
clog, 9–6
conj, 9–8
cos, 9–17
cosh, 9–17
cpow, 9–6
cproj, 9–8
creal, 9–8
csin, 9–4
Library function (cont'd)
csinh, 9–5
csqrt, 9–7
ctan, 9–4
ctanh, 9–6
ctime, 9–51
difftime, 9–53
div, 9–40
exit, 9–38
exp, 9–18
fabs, 9–19
fclose, 9–26
feof, 9–34
ferror, 9–34
fflush, 9–27
fgetc, 9–30
fgetpos, 9–33
fgets, 9–30
floor, 9–19
fmod, 9–19
fopen, 9–27
fprintf, 9–28
fputc, 9–30
fputs, 9–31
fread, 9–32
free, 9–37
freopen, 9–27
frexp, 9–18
fscanf, 9–29
fseek, 9–33
fsetpos, 9–33
ftell, 9–34
fwrite, 9–32
getc, 9–31
getchar, 9–31
getenv, 9–38
gets, 9–31
gmtime, 9–51
isalnum, 9–9
isalpha, 9–9
iscntrl, 9–9
isdigit, 9–9
isgraph, 9–9
islower, 9–9
isprint, 9–9

Library function (cont'd)
ispunct, 9–9
isspace, 9–9
isupper, 9–10
isxdigit, 9–10
labs, 9–40
ldexp, 9–18
ldiv, 9–40
localeconv, 9–13
localtime, 9–51
log, 9–18
log10, 9–18
longjmp, 9–20
malloc, 9–37
mblen, 9–40
mbstowcs, 9–41
mbtowc, 9–40
memchr, 9–43
memcmp, 9–43
memcpy, 9–43
memmove, 9–43
memset, 9–43
mktime, 9–53
modf, 9–18
perror, 9–34
pow, 9–18
printf, 9–29
putc, 9–31
putchar, 9–31
puts, 9–32
qsort, 9–39
raise, 9–21
rand, 9–37
realloc, 9–37
remove, 9–25
rename, 9–26
rewind, 9–34
scanf, 9–29
setbuf, 9–28
setjmp, 9–19
setlocale, 9–12
signal, 9–20
sin, 9–17
sinh, 9–17
sprintf, 9–29
Library function (cont'd)
sqrt, 9–18
srand, 9–37
sscanf, 9–29
strcat, 9–44
strchr, 9–45
strcmp, 9–44
strcoll, 9–44
strcpy, 9–43
strcspn, 9–45
strerror, 9–46
strftime, 9–51
strlen, 9–46
strncat, 9–44
strncmp, 9–44
strncpy, 9–43
strpbrk, 9–45
strrchr, 9–45
strspn, 9–45
strstr, 9–45
strtod, 9–36
strtok, 9–45
strtol, 9–36
strtoul, 9–36
strxfrm, 9–44
system, 9–38
tan, 9–17
tanh, 9–17
time, 9–53
tmpfile, 9–26
tmpnam, 9–26
tolower, 9–10
toupper, 9–10
ungetc, 9–32
vfprintf, 9–29
vprintf, 9–30
vsprintf, 9–30
wcstombs, 9–42
wctomb, 9–41
Limit of nested #include lines, 8–11
Limits
ANSI, 1–24
defined, 1–24
header file, 9–11
numerical, 1–25

Limits (cont'd)
translation, 1–25
<limits.h> header, 9–11
__LINE__ predefined macro, 8–16
#line preprocessing directives, 8–12
Linkage
determination, 2–12
example, 2–12
external, 2–11
internal, 2–11
none, 2–11
type, 2–11
Linkage pragma, B–17
Literal
compound, 6–29
<locale.h> header, 9–11
localeconv library function, 9–13
Localization
header file, 9–11
localtime library function, 9–51
log10 library function, 9–18
Logical
arithmetic operator, 6–24
negation operator, 6–13
&& (logical AND operator), 6–24
!(logical expression), 6–13
Logical false, definition, 7–4
Logical line, 1–4
|| (logical OR operator), 6–24
Logical true, definition, 7–4
log library function, 9–18
long float keyword, D–1
longjmp library function, 9–20
long keyword, 4–10
Looping statement
See Iteration statement
L preceding wide character, 1–20
lvalue, 2–25
L_tmpnam macro, 9–25

M

-- (postfix decrement operator), 6–11
Macro
assert, 9–2
bool, 9–23
__bool_true_false_are_defined, 9–23
BUFFSIZ, 9–24
CLOCKS_PER_SEC, 9–50
complex, 9–3
_Complex_I, 9–3
definition, 8–2
canceling, 8–2
function-like form, 8–4
naming parameters in, 8–4
object-like form, 8–4
operator, 8–7
operator, 8–8
possible side effects, 8–6
EDOM, 9–10
EOF, 9–24
ERANGE, 9–10
errno, 9–10
EXIT_FAILURE, 9–35
EXIT_SUCCESS, 9–35
false
stdbool.h, 9–23
FILENAME_MAX, 9–25
FOPEN_MAX, 9–25
HUGE_VAL, 9–16
imaginary, 9–3
_Imaginary_I, 9–3
INFINITY, 9–16
_IOFBF, 9–24
_IOLBF, 9–24
_IONBF, 9–24
LC_ALL, 9–11
LC_COLLATE, 9–11
LC_MONETARY, 9–11
LC_NUMERIC, 9–11
LC_TIME, 9–11
LC_TYPE, 9–11
library, 9–1

Macro (cont'd)
L_tmpnam, 9–25
MB_CUR_MAX, 9–36
NAN, 9–16
NULL
limits.h, 9–11
stddef.h, 9–23
stdio.h, 9–24
stdlib.h, 9–35
string.h, 9–42
time.h, 9–50
offsetof, 9–23
predefined
__DATE__, 8–16
defined, 8–16
__FILE__, 8–16
__LINE__, 8–16
__STDC__, 8–17
__STDC_HOSTED__, 8–17
__STDC_ISO_10646__, 8–17
__STDC_VERSION__, 8–17
system identification, 8–17
__TIME__, 8–17
RAND_MAX, 9–35
references, 8–6
SEEK_CUR, 9–25
SEEK_END, 9–25
SEEK_SET, 9–25
SIG_DFL, 9–20
SIG_ERR, 9–20
SIG_IGN, 9–20
stderr, 9–25
stdin, 9–25
stdout, 9–25
substitution, 8–2
substitution within #include directives, 8–12
TMP_MAX, 9–25
truestdbool.h, 9–23
va_arg, 9–22
va_list, 9–22
va_start, 9–22
Macro expansion, in pragmas, 8–13
Macro names
list, B–13
Main function
passing parameter, 5–14
syntax, 5–14
with main_program option, E–2
main_program option, E–2
malloc library function, 9–37
Math, type-generic
header file, 9–46
<math.h> header, 9–16
Mathematics
complex, 9–2
header file, 9–16
mblen library function, 9–40
mbstowcs library function, 9–41
mbtowc library function, 9–40
MB_CUR_MAX macro, 9–36
Member
structure, B–7
union, B–7
variant aggregate, E–4
memchr library function, 9–43
memcmp library function, 9–43
memcpy library function, 9–43
memmove library function, 9–43
memset library function, 9–43
Minus
unary, 6–13
mktime library function, 9–53
modf library function, 9–18
Modifiable lvalue, 2–26
Modifier
storage class, 2–16
#module directive, E–1
Multibyte character
defined, 1–21
HP C, B–2
Multidimensional array, 4–18
subscripts, 3–13
* (multiplication operator), 6–20
Multiplicative operator, B–8
Multiplicative operator (*), 6–20

N

Name space
  defined, 2–26
  types, 2–26
NAN macro, 9–16
Negation
  arithmetic, 6–13
  logical, 6–13
Nesting of #include lines, 8–11
New-line character, 1–4
New style parameter declaration, 5–5
Nonlocal jump
  header file, 9–19
Notational conventions, xvii
Not-equal-to operator (!=), 6–23
null (#) preprocessing directive, 8–16
Null character, 1–4
NULL macro
  limits.h, 9–11
  stddef.h, 9–23
  stdio.h, 9–24
  stdlib.h, 9–35
  string.h, 9–42
  time.h, 9–50
Null pointer
  automatic initialization, 4–14
  defined, 3–12
  used with the equality operator, 6–27
Null statement, 7–3
Numerical limit
  defined, 1–25
Numerical limits, B–3
Numeric escape sequence, 1–22 to 1–23

O

Octal constant, 1–14
0...Octal constant, 1–14
Octal digits 8 and 9, B–5, D–1
offsetof macro, 9–23
Old-style function declaration
  combined with prototype style, 2–10
Old style parameter declaration
  comparison with prototype style, 5–8
  example, 5–5
One's complement operator (~), 6–15
Operand conversion, 6–33
operator, B–15
Operator
  assignment, 6–26 to 6–27
  binary
    additive, 6–21
    bitwise, 6–24
    equality, 6–23
    logical, 6–24
    multiplicative, 6–20
    relational, 6–23
    shift, 6–22
  bracket, 6–7
  categories, 6–4
  comma, 6–27
  conditional, 6–25
  defined, 1–11
  defined, 8–11
  list, 6–3
  _Pragma, 6–19
  precedence, 6–5
  sizeof, 6–17
  __typeof__, 6–18
  unary
    address, 6–15
    arithmetic negation, 6–13
    bitwise negation, 6–15
    cast, 6–15
    increment and decrement, 6–13
    indirection, 6–15
    logical negation, 6–13
OR bitwise operator (|), 6–24
Original declaration, 4–8

P

Parameter
  function prototype, 5–8
  in #define preprocessor macros, 8–4
  main function, 5–15
  rules governing, 5–11

Parameter passing
to main function, 5–14
Parenthesized expression, 6–2
( ) (Parenthesized expression), 6–2
Parsing
top-down, 4–7
Parsing error, 1–2
Period operator (.), 6–10
perror library function, 9–34
Pointer
declaration, 4–13
null, 4–14
to array, 4–21
unary operator, 6–15
using the increment operator (++), 6–14
void, 4–14
Pointer conversion, B–7
Pointer declaration
syntax, 4–13
Pointer initialization, 4–15
Pointer to void, 3–12
Pointer type
discussed, 3–12
referenced type, 3–12
Portability concern
main_program option, E–2
Portability concerns
char * generic-pointer notation, 4–14
length of bit field, 4–28
preprocessor implementations, 8–1
structure alignment, 4–27
Postfix decrement operator, 6–11
Postfix expression
array reference, 6–7
decrement operator, 6–11
function call, 6–8
increment operator, 6–11
structure reference, 6–10
union reference, 6–10
Postfix increment operator, 6–11
++ (postfix increment operator), 6–11
pow library function, 9–18
Pragma
directive, B–16
VAX C, B–17
_Pragma operator, 6–19
#pragma preprocessing directive, 8–13
Precedence
defined, 1–11
discussed, 6–5
operator, 6–5
Predefined alignment constants, 2–22
Predefined macro names, 8–16
Prefix decrement operator, 6–13
-- (prefix decrement operator), 6–13
Prefix increment operator, 6–13
++ (prefix increment operator), 6–13
Preprocessing
discussed, 2–27
Preprocessing operator
#, 8–7
##, 8–8
Preprocessor directive
#define, 8–2
#elif, 8–8
#else, 8–8
#endif, 8–8
#error, 8–16
#if, 8–8
#ifdef, 8–8
#ifndef, 8–8
#include
defined, 8–11
macro substitution, 8–12
#line, 8–12
null (#), 8–16
#pragma, 8–13
#undef, 8–2
Primary expression
constant, 6–2
defined, 6–2
identifier, 6–2
parentheses, 6–2
Primary operator
precedence of, 6–5
printf library function, 9–29
Promotion of data type, 6–32
Prototype
defined, 5–8
for C library function, 5–10

Prototype style function declaration
combined with old-style, 2–10
defined, 5–8
Prototype style parameter declaration, 5–5
ptrdiff type, 9–23
... punctuator, 5–9
Punctuator, 1–12 to 1–13
putchar library function, 9–31
putc library function, 9–31
puts library function, 9–32

Q

qsort library function, 9–39

R

raise library function, 9–21
rand library function, 9–37
RAND_MAX macro, 9–35
Reader's comments, xiii
realloc library function, 9–37
register class, 2–15
register keyword
used in declaration inside block, 7–2
Relational operator, 6–23
Relaxed pointer/integer compatibility
common C, D–1
VAX C, E–1
Remainder operator (%), 6–20
% (remainder operator), 6–20
remove library function, 9–25
rename library function, 9–26
__restrict data-type qualifier, 4–14
Restricted pointer, 3–24
defined, 3–27
examples, 3–29
__restrict type qualifier, 3–24
defined, 3–27
examples, 3–29
return keyword
statement syntax, 7–12
rewind library function, 9–34
>> (right shift operator), 6–22
Routine
library, 9–1
rvalue, 2–25

S

Scaling pointer, 4–21
scanf library function, 9–29
Scope, 2–3
SEEK_CUR macro, 9–25
SEEK_END macro, 9–25
SEEK_SET macro, 9–25
Selection statement, 7–4 to 7–8
Semantic error, 1–2
Sequence point, 2–7
setbuf library function, 9–28
<setjmp.h> header, 9–19
setjmp library function, 9–19
setlocale library function, 9–12
Shift operator (<< and >>), 6–22
Side effect
defined, 2–6
example, 2–7
within macros, 8–6
SIGABRT signal, 9–21
SIGFPE signal, 9–21
SIGILL signal, 9–21
<signal.h> header, 9–20
Signal handling
header file, 9–20
signal library function, 9–20
Signed integer, B–6
Significant characters
identifier, 1–7
SIGSEGV signal, 9–21
SIGTERM signal, 9–21
sig_atomic_t type, 9–20
SIG_DFL macro, 9–20
SIG_ERR macro, 9–20
SIG_IGN macro, 9–20
Similarity between structure and union, 4–26

Simple object
declaration format, 4–9
default initialization, 4–10
initialization, 4–10
sinh library function, 9–17
sin library function, 9–17
sizeof keyword, 6–17
sizeof operator, 6–17, B–8
size_t type
stddef.h, 9–23
stdio.h, 9–24
stdlib.h, 9–35
string.h, 9–42
time.h, 9–50
Source character set
defined, 1–3
storage, 3–8
Source file inclusion, B–13
sprintf library function, 9–29
sqrt library function, 9–18
srand library function, 9–37
sscanf library function, 9–29
Standard header file, 9–1
Statement
break
defined, 7–12
from case statement, 7–5
compound, or block, 7–2
conditional, 7–4 to 7–8
continue, 7–11
default, 7–5
do
defined, 7–9
iteration statement, 7–8
expression, 7–3
for
defined, 7–9
iteration statement, 7–8
goto, 7–11
if, 7–4
iteration, 7–8 to 7–10
jump, 7–11 to 7–12
label, 7–1
like, 7–8
null, 7–3
Statement (cont'd)
return, 7–12
selection, 7–4 to 7–8
switch, 7–5
while, 7–8
static class, 2–15
static keyword
used in declaration inside block, 7–2
Static storage duration, 2–13
<stdarg.h> header, 9–21
<stdbool.h> header, 9–22
__STDC__ predefined macro, 8–17
__STDC_HOSTED__ predefined macro, 8–17
__STDC_ISO_10646__ predefined macro, 8–17
__STDC_VERSION__ predefined macro, 8–17
<stddef.h> header, 9–23
stderr macro, 9–25
stdin macro, 9–25
<stdio.h> header, 9–24
<stdlib.h> header, 9–35
stdout macro, 9–25
Storage allocation
order, 4–4
to an object, 4–4
Storage class
default, 2–14
types, 2–13
Storage classes
auto, 2–14
extern, 2–15
register, 2–15
static, 2–15
Storage-class modifiers, 2–16
__align, 2–21
__forceinline, 2–21
inline, 2–17
__inline, 2–17
Storage class specifier, B–9
Storage duration, 2–13
strcat library function, 9–44

strchr library function, 9–45
strcmp library function, 9–44
strcoll library function, 9–44
strcpy library function, 9–43
strcspn library function, 9–45
strerror library function, 9–46
strftime library function, 9–51
String
  defined, 1–13
<string.h> header, 9–42
String literal
  defined, 1–13
  example, 1–13
  HP C, B–5
  modifying, 4–15
String processing
  header file, 9–42
strlen library function, 9–46
strncat library function, 9–44
strncmp library function, 9–44
strncpy library function, 9–43
strpbrk library function, 9–45
strrchr library function, 9–45
strspn library function, 9–45
strstr library function, 9–45
strtod library function, 9–36
strtok library function, 9–45
strtol library function, 9–36
strtoul library function, 9–36
struct lconv type, 9–11
struct tm type, 9–50
Structure
  bit field, 4–28
  declaration, 4–24 to 4–26
  forward referencing, 4–26
  initialization, 4–30, 4–33
  member
    reference, 6–10
  variant aggregate, E–4
Structure alignment
  described, B–10
  discussed, 3–15
  on OpenVMS Alpha, B–10
  on OpenVMS VAX, B–11
  on Tru64 UNIX, B–10
. (structure and union operator), 6–10
Structure declaration
  syntax, 4–23, 4–24
Structure member
  declaration, 4–24
-> (structure or union pointer operator), 6–10
Structure specifier, B–9
Structure type, 3–13
strxfrm library function, 9–44
Subscripting, 6–7
Substitution
  macro, 8–2
    rules, 8–5
  within #include directives, 8–12
switch keyword
  declaration inside, 7–7
switch statement, 7–5
  HP C behavior, B–12
Syntax
  main function, 5–14
Syntax error, 1–2
System identification predefined macros, 8–17
system library function, 9–38

T

Tag
  declaration syntax, 4–36
  discussed, 2–24
  example, 2–24
  tentative declaration, 4–26
tanh library function, 9–17
tan library function, 9–17
Tentative definition
  defined, 4–8
  discussed, 2–13
  example, 2–13
Tentative tag declaration, 4–26
Ternary operator, 1–11
<tgmath.h> header, 9–46
<time.h> header, 9–50

__TIME__ predefined macro, 8–17
Time and Date
  header file, 9–50
time library function, 9–53
tmpfile library function, 9–26
tmpnam library function, 9–26
TMP_MAX macro, 9–25
Token, 1–1
Token replacement, 8–2
tolower library function, 9–10
toupper library function, 9–10
_toupper macro
  function-like form, 8–4
  side effects, 8–6
Translation
  of C code, 2–27
  phase, 2–27
Translation limit, 1–25
Translation limits, B–2
Trigraph sequence, 1–5
true macro
  stdbool.h, 9–23
Type
  clock_t, 9–50
  div_t, 9–35
  FILE, 9–24
  fpos_t, 9–24
  HP C, B–5
  incomplete, 2–8
  jmp_buf, 9–19
  ldiv_t, 9–35
  library, 9–1
  ptrdiff, 9–23
  sig_atomic_t, 9–20
  size_t
    stddef.h, 9–23
    stdio.h, 9–24
    stdlib.h, 9–35
    string.h, 9–42
    time.h, 9–50
  struct lconv, 9–11
  struct tm, 9–50
  va_list, 9–21
  wchar_t
    stddef.h, 9–23
Type
  wchar_t (cont'd)
    stdlib.h, 9–35
Type casting, 6–15
Type conversion, 6–15
typedef
  and structure tags, 4–37
  defined, 3–35
Type definition, 3–35 to 3–36
Type definitions, 4–37
typedef keyword
  use in declaration, 4–37
Type-generic math
  header file, 9–46
Type name, 2–28
__typeof__ operator, 6–18
type qualifier
  const, 3–20
  __restrict, 3–24
    defined, 3–27
    examples, 3–29
  __unaligned, 3–23
  volatile, 3–22
Type qualifier, 3–19 to 3–35
  defined, 3–19
  HP C, B–11
  use, 3–19
Type specifier, B–9

U

Unaligned data, 3–23
__unaligned data-type qualifier, 4–14
__unaligned type qualifier, 3–23
Unary expression
  address, 6–15
  arithmetic negation, 6–13
  bitwise negation, 6–15
  cast, 6–15
  increment and decrement, 6–13
  indirection, 6–15
  logical negation, 6–13
  _Pragma, 6–19
  sizeof, 6–17
  __typeof__, 6–18

Unary minus, 6–13
Unary operator
  defined, 1–11
  precedence of, 6–5
#undef preprocessing directive, 8–2
ungetc library function, 9–32
Union
  declaration, 4–24
  initialization, 4–32
  member
    reference, 6–10
  variant aggregate, E–4
Union declaration
  syntax, 4–23, 4–24
Union specifier, B–9
Union type
  discussed, 3–15
Universal character names, 1–8, F–1
Unnamed bit field, 4–29
Unsigned integer, B–6
Unsigned integral type, 3–7
Unsigned preservation rules
  Common C mode, D–3
  VAX C mode, E–3
User-defined function
  See Function
Usual Arithmetic conversions, 6–33
Utilities
  header file, 9–35

V

Vacuous tag declaration
  example, 2–25
Variable
  initialization, 4–10
Variable arguments
  header file, 9–21
Variable-length array, 4–16, 4–22
Variable-length parameter list, 5–9
Variably modified type, 4–2
Variant record, 3–15
Variant structure
  defined, E–4
  HP C behavior, B–9
Variant union, E–4
  HP C behavior, B–9
variant_struct, E–4
variant_union, E–4
VAX C built-in functions, E–1
VAX C extension of HP C, E–1
VAX C keywords, E–1
va_arg macro, 9–22
__VA_ARGS__ identifier, 8–4, 8–5
va_list macro, 9–22
va_list type, 9–21
va_start macro, 9–22
vfprintf library function, 9–29
Visibility
  defined, 2–6
  discussed, 2–6
  example, 2–6
void keyword
  pointer, 4–14
void pointer
  defined, 4–14
  uses, 3–12
Void type
  defined, 3–17
  example, 3–17
  use, 3–17
volatile type qualifier
  discussed, 3–22
  rules, 3–22
vprintf library function, 9–30
vsprintf library function, 9–30

W

wchar_t type
  stddef.h, 9–23
  stdlib.h, 9–35
wcstombs library function, 9–42
wctomb library function, 9–41
while statement, 7–8
White space
  character, 1–4
  use, 1–2

Wide character
defined, 1–20
use, 1–3
Wide-character constant
defined, 1–20
example, 1–20
Wide-character string, 1–20
Wide character type, 3–9

X

0x...Hexadecimal constant, 1–14
XOR bitwise operator (^), 6–24

Z

Zero-length bit field, 4–29