HP C/HP-UX Reference Manual

Version A.06.05

Edition 5

Manufacturing Part Number: B3901-90020

Septeember 2005

Printed in the U.S.A.

Legal Notices

Contents

About This Document

This manual presents reference information on the C programming language, as implemented on Itanium®- based systems.

The document printing date and part number indicate the document's current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. The document part number will change when extensive changes are made.

Document updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. Contact your HP sales representative for details.

The latest version of this document can be found on line at `http://docs.hp.com`.

Intended Audience

This manual is intended for experienced C programmers who are familiar with HP computer systems.

Printing History

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

Table 1 Printing History

Edition	Release Date	Product Version
Edition 1	August 2003	HP aC++ v A.05.50
Edition 2	March 2004	HP aC++ v A.05.55
Edition 3	September 2004	HP aC++ v A.05.55.02
Edition 4	December 2004	HP aC++ v A.06.00/A.05.60
Edition 5	September 2005	HP aC++ v A.06.05

Related Documents

The following is a list of documents available with this release:

- *HP aC++/HP ANSI C Release Notes*
- *HP aC++/HP C Programmer's Guide*
- *HP C/HP-UX Reference Manual*

Additional information about the HP C compiler can be found at `http://www.docs.hp.com` in the *Development Tools and Distributed Computing* section.

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

- *American National Standard for Information Systems — Programming Language — C, ANSI/ISO 9899-1990.*
- *COBOL/HP-UX Operating Manual* — This manual provides information on calling C subprograms from COBOL programs on HP-UX. It also explains how to call COBOL subprograms from C.
- *HP-UX 64-bit Porting and Transition Guide* — Describes the changes you need to make to compile, link, and run programs in 64-bit mode. This document is also available online at http://docs.hp.com, and in the Postscript file `/opt/ansic/newconfig/RelNotes/64bitTrans.bk.ps`.
- *HP-UX Floating-Point Guide* — This manual describes the IEEE floating-point standard, the HP-UX math libraries on HP 9000 systems, performance tuning related to floating-point routines, and floating-point coding techniques that can affect application results.
- *HP Fortran 90 Programmer's Guide* — This manual explains how to call C programs from the HP Fortran 90 compiler on HP-UX.
- *HP Pascal/HP-UX Programmer's Guide* — This manual describes how to call C programs from Pascal on HP-UX systems.
- *HP-UX Linker and Libraries Online User Guide* — This online help describes programming in general on HP-UX. For example, it covers linking, loading, shared libraries, and several other HP-UX programming features.
- *HP-UX Reference* — Manpages for HP-UX 11.00 are available in Instant Information under the title *HP-UX Reference* and via the `man` command. They document commands, system calls, subroutine libraries, file formats, device files, and other HP-UX related topics.
- *Parallel Programming Guide for HP-UX Systems* — Describes efficient parallel programming techniques available using HP Fortran 90, HP C, and HP aC++ on HP-UX.

HP Encourages Your Comments

HP encourages your comments concerning this document. We are truly committed to providing documentation that meets your needs.

Please send comments to: `c++-editor@cup.hp.com`

Please include document title, manufacturing part number, and any comment, error found, or suggestion for improvement you have concerning this document.

1 What is HP C?

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

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

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

ANSI Mode

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

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

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

Compatibility Mode

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

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

About HP C/HP-UX

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

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

HP C Online Help

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

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

Prerequisites for using HP C Online Help

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

- You must set the `DISPLAY` environment variable to a (graphical mode) value that can accommodate the display of an HTML browser.
- You may set the `BROWSER` environment variable to point to the location of the your HTML browser. If you do not do this, the compiler will automatically run the browser located in `/opt/ns-navgold/bin/netscape` or in `/opt/ns-communicator/netscape`.
- You may set the `CROOTDIR` environment variable to specify the root directory of the online help source. If `CROOTDIR` is not set, the URL of the HP C Online Help defaults to `file:/opt/ansic/html/guide/${LOCALE}/c_index.html`. This default is based on the assumption that the compiler binaries are located in `/opt/ansic/bin`.

Accessing HP C Online Help

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

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

The actual location of the HTML files is:

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

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

NOTE If the browser path set by the `BROWSER` environment variable does not exist, or if the default browser paths `/opt/ns-navgold/bin/netscape` or `/opt/ns-communicator/netscape` do not exist, then you must set the `BROWSER` environment variable appropriately.

2 Program Organization

This section describes the following topics:

Lexical Elements

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

- White Space, Newlines, and Continuation Lines
- Spreading Source Code Across Multiple Lines
- Comments
- Identifiers
- Keywords

White Space, Newlines, and Continuation Lines

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

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

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

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

Spreading Source Code Across Multiple Lines

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

```
#define foo_macro(x,y,z) ((x) + (y))\
                         * ((z) - (x))

printf("This is an very, very, very lengthy and \
very, very uninteresting string.");
```

Comments

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

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

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

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

Comments may also span multiple lines, as in:

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

Identifiers

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

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

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

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

Identifiers cannot conflict with reserved Keywords.

Legal Identifiers

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

Illegal Identifiers

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

Length of Identifiers

HP C identifiers are unique up to 256 characters.

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

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

Case Sensitivity in Identifiers

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

```
kilograms
KILOGRAMS
Kilograms
```

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

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

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

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

Predefined identifier _func_

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

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

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

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

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

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

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

```
varfunc
```

Keywords

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

- `auto`
- `break`
- `case`
- `char`
- `const`
- `continue`
- `default`
- `do`
- `double`
- `else`

- `enum`
- `extern`
- `float`
- `for`
- `goto`
- `if`
- `int`
- `long`
- `register`
- `return`
- `short`
- `signed`
- `sizeof`
- `static`
- `struct`
- `switch`
- `__thread`
- `typedef`
- `union`
- `unsigned`
- `void`
- `volatile`
- `while`

auto

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

break

See “break” on page 156.

case

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

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

char

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

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

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

const

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

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

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

continue

See “continue” on page 157.

default

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

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

do

See “do…while” on page 159.

double

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

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

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

else

See “if” on page 169.

enum

See “Enumeration” on page 52.

extern

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

float

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

The range for `float` is:

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

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

for

See “for” on page 161.

goto

See “goto” on page 166.

if

See “if” on page 169.

int

A 32-bit data type for representing whole numbers.

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

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

long

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

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

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

register

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

return

See “return” on page 174.

short

A 16-bit integer data type.

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

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

signed

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

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

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

sizeof

See “sizeof Operator” on page 131.

static

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

struct

See “Structure and Union Tags” on page 49.

switch

See “switch” on page 178.

__thread

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

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

```
__thread int j=2;
int main()
{
   j = 20;
}
```

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

Only one declaration, for example,

```
__thread int x;
```

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

```
extern __thread int x;
```

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

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

typedef

See “Typedef Declarations” on page 19.

union

See “Structure and Union Tags” on page 49.

unsigned

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

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

void

The `void` data type has three important purposes:

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

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

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

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

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

volatile

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

while

See “while” on page 182.

Declarations

In general, a variable declaration has the following format:

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

where:

storage_class_specifier	is an optional keyword.
data_type	is one of the data types described in Chapter 3, “Data Types and Declarations.”
variable_name	is a legal identifier.

initial_value is an optional initializer for the variable.

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

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

Typedef Declarations

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

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

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

The following statement makes the name `FOUR_BYTE_INT` synonymous with `long int`:

```
typedef long int FOUR_BYTE_INT;
```

The following two declarations are now identical:

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

Abstract Global Types

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

```
typedef struct {
    char month[4];
    int day;
    int year;
} BIRTHDAY;

typedef char A_LINE[80]; /* A_LINE is an array of */
                        /* 80 characters        */
```

Improving Portability

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

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

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

Simplifying Complex Declarations

You can use `typedef`s to simplify complex declarations. For example:

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

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

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

Using typedefs for Arrays

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

Table 2-1 Mixing Pointers and Typedefs

Wrong	Right
`typedef char STR[80];`	`typedef char STR[80];`
`STR   string, *ptr;`	`STR   string;`
	`char *ptr;`
`main()`	`main()`
`{`	`{`
`   ptr = string;`	`   ptr = string;`
`   printf("ptr = %d\n", ptr);`	`   printf("ptr = %d\n", ptr);`
`   ptr++;`	`   ptr++;`
`   printf("ptr = %d\n", ptr);`	`   printf("ptr = %d\n", ptr);`
`}`	`}`
`*** Run-Time Results ***`	`*** Run-Time Results ***`
`ptr = 3997696`	`ptr = 3997696`
`ptr = 3997776`	`ptr = 3997697`

Name Spaces

All identifiers (names) in a program fall into one of four name spaces. Names in different name spaces never interfere with each other. That is, you can use the same name for an object in each of the four name spaces without these names affecting one another. Table 2-2 lists the four name spaces:

Table 2-2 Name Spaces

Name Spaces	Description
Structure, Union, and Enumeration Tags	Tag names that immediately follow these type specifiers: `struct`, `union`, and `enum`. These types are described in “Structure and Union Specifiers” on page 48.
Member Names	Names of members of a structure or union.

Table 2-2 Name Spaces (Continued)

Name Spaces	Description
Goto Labels	Names that mark the target of a goto statement.
Function, Variable and All Other Names	Any name that is not a member of the preceding three classes.

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

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

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

Structure, Union, and Enum Names

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

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

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

Macro Names

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

```
#define square(arg)  arg * arg

int main(void)
{
   ...
   square:
   ...
}
```

Declarations within code

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

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

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

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

Constants

There are four types of constants in C:

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

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

Integer Constants

HP C supports three forms of integer constants:

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

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

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

Table 2-3 Convention Summary

Constant	Assigned Data Type
decimal (no suffix)	`int`, `long int`, `unsigned long int`
octal or hex (no suffix)	`int`, `unsigned int`, `long`, `unsigned long`
letter `u` or `U` suffix	`unsigned int`, `unsigned long int`
letter `l` or `L` suffix	`long`, `unsigned long`
both letters `u` or `U` and	`unsigned long l` or `L` suffix
letters `ll` or `LL` suffix:	`long long`, `unsigned long long`

Table 2-3 Convention Summary (Continued)

Constant	Assigned Data Type
both letters `u` or `U` and `ll` or `LL` suffix:	unsigned long long

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

Examples of Integer Constants

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

Table 2-4 Convention Summary

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

Floating-Point Constants

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

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

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

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

Scientific Notation

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

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

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

Examples of Floating-Point Constants

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

Table 2-5 Floating-Point Constants

Constant	Legal or Illegal
3.	legal
35	legal — interpreted as an integer.
3.141	legal
3,500.45	illegal — commas are illegal.
.3333333333	legal
4E	illegal — the exponent must be followed by a number
0.3	legal
-3e2	legal
4e3.6	illegal — the exponent must be an integer
3.0E5	legal
+3.6	legal
0.4E-5	legal

Character Constants

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

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

Escape Sequences

HP C supports several escape sequences:

Table 2-6 Character Escape Codes

Escape Code	Character	What it Does
`\a`	Audible alert	Rings the terminal's bell.
`\b`	Backspace	Moves the cursor back one space.
`\f`	Form feed	Moves the cursor to the next logical page.
`\n`	Newline	Prints a newline.
`\r`	Carriage return	Prints a carriage return.
`\t`	Horizontal tab	Prints a horizontal tab.
`\v`	Vertical tab	Prints a vertical tab.
`\\`	Backslash	Prints a backslash.
`\?`	Question mark	Prints a question mark.
`\'`	Single quote	Prints a single quote.
`\"`	Double quote	Prints a double quote.

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

Multi-Character Constants

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

For example, the following assignments are legal:

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

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

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

The variable `si` is assigned the value of `e` and `f`, where each character takes up 8 bits of the 16-bit value. The HP C compiler places the last character in the rightmost (least significant) byte. Therefore, the constant `ef` will have a hexadecimal value of 6566. Since the order in which bytes are assigned is machine dependent, other machines may reverse the order, assigning `f` to the most significant byte. In that case, the resulting value would be 6665. For maximum portability, do not use multi-character constants. Use character arrays instead.

String Constants

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

```
"A short string"
```

becomes an array with 15 elements:

Figure 2-1 String Constants

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

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

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

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

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

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

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

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

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

Structuring a C Program

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

- Preprocessor directives
- Variables
- Functions

Example 2-1 Example

The following shows how a program can be organized:

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

/* global typedef declaration */
typedef float THIRTY_TWO_BIT_REAL;

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

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

/* start of function body */
{

/* local variable declaration */
   float mean;

/* assignment statement */
   mean = (arg1 * WEIGHTING_FACTOR) + (arg2 * (1.0 - WEIGHTING_FACTOR));

/* return statement */
return (mean * correction_factor);

/* end of function body */

}

int main(void)
```

```
   /* start of function body */
{

   /* local variable declarations */
   float value1, value2, result;

   /* statements */
   printf("Enter two values -- ");
   scanf("%f%f", &value1, &value2);
   result = average(value1, value2);

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

   /* end of function body */
}
```

3 Data Types and Declarations

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

The declarable entities in C are:

- Objects
- Functions
- Tags and members of structures, unions, and enumerated types
- Type definition names

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

Program Structure

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

Syntax

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

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

Description

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

Declarations

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

Syntax

```
declaration ::=
    declaration-specifiers [init-declarator-list] ;

declaration-specifiers ::=
     storage-class-specifier [declaration-specifiers]
     type-specifier [declaration-specifiers]
     type-qualifier [declaration-specifiers]

init-declarator-list ::=
     init-declarator
     init-declarator-list , init-declarator

init-declarator ::=
     declarator
     declarator = initializer
```

Description

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

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

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

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

New Declaration Features

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

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

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

Caveats

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

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

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

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

Example

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

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

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

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

Examples

Valid Declarations:

```
extern int pressure [ ];          /* size will be declared elsewhere */
extern int lines = 66, pages;     /* declares two variables,
                                     initializes the first one */
static char private_func (float); /* a function taking a float,
                                     returning a char, not known
                                     outside this unit */
const float pi = 3.14;            /* a constant float, initialized */

const float * const pi_ptr = &pi; /* a constant pointer to a constant
                                     float, initialized with an
                                     address constant */
static j1, j2, j3;                 /* initialized to zero by default */
typedef struct {double  real, imaginary;} Complex;
                                   /* declares a type name */
Complex impedance = {47000}; /* second member defaults to zero */
enum color {red=1, green, blue};   /* declares an enumeration tag and
                                      three constants */
int const short static volatile signed
 really_Strange = {sizeof '\?'};   /* pretty mixed up */
```

Invalid Declarations:

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

Storage-Class Specifiers

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

Syntax

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

Description

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

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

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

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

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

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

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

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

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

Type Specifiers

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

Syntax

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

Description

Most of the type specifiers are single keywords. (Refer to Chapter 10, "HP C/HP-UX Implementation Topics," on page 235 for sizes of types.) The syntax of the type specifiers permits more types than are actually allowed in the C language. The various combinations of type specifiers that are allowed are shown in Table 3-1. Type specifiers that are equivalent appear together in a box. For example, specifying `unsigned` is equivalent to `unsigned int`. Type specifiers may appear in any order, possibly intermixed with other declaration specifiers.

Table 3-1 C Type Specifiers

`void`
`char`
`signed char`

Table 3-1 C Type Specifiers (Continued)

`unsigned char`
`short`, `signed short`, `short int`, or `signed short int`
`unsigned short`, or `unsigned short int`
`int`, `signed`, `signed int`, or *no type specifiers*
`unsigned`, or `unsigned int`
`long`, `signed long`, `long int`, or `signed long int`
`long long`, `signed long long`, `long long int`, or `signed long long int`
`unsigned long`, or `unsigned long int`
`unsigned long long`, or `unsigned long long int`
`float`
`double`
`long double`
`_Bool`
`float_Complex`
`double_Complex`
`long double_Complex`
`float_Imaginary`
`double_Imaginary`
`long double_Imaginary`
struct-or-union specifier
enum-specifier
typedef-name

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

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

_Bool

This release supports the boolean data type `_Bool`. The variables of the data type `_Bool` can only have values true(1) or false(0), where `true` and `false` are preprocessor macros defined in the header file `stdbool.h`. The _Bool data type is a part of C99 standard (ISO/IEC 9899:1999). The C99 standard specifies boolean bitfields. For example: _Bool can be defined as in the following structure declaration:

```
struct foo {
            _Bool boolval:1;
            int i;
           }
```

Since _Bool is defined to take only 0 and 1 as values, this type of _Bool declaration has some special properties, such as:

```
_Bool flip_flop = 0; // flip_flop is now false
++flip_flop;         // flip_flop is now true
++flip_flop;         // flip_flop is true
--flip_flop;         // flip_flop is now false
--flip_flop;         // flip_flop is now true
```

New Header file

The default location for `<stdbool.h>` is `/usr/include`. This header file is available in patch PHSS_24204 (for HP-UX 11.00) and PHSS_24205 (for HP-UX 11.11). This header file includes the following four macros:

- *bool* expands to _Bool.
- *true* expands to integer constant 1.
- *false* expands to integer constant 0.
- *__bool_true_false_are_defined* expands to decimal constant 1.

The last three macros are suitable for use in `#if` preprocessor directives.

Usage of _Bool

Observe the following while using `_Bool`:

- The rank of `_Bool` is less than the rank of all other standard integer types.

- A bit field of type `_Bool` may be used in an expression where an `int` or `unsigned int` is used.
- When a scalar value is converted to `_Bool` the result is 0, if the value compares equal to 0, else the result is 1.

Rules for _Bool Conversion

The following conversion rules are applicable while using _Bool.

Scalar to _Bool

```
Boolval = scalarval ? true : false;
```

Boolval would be true(1) or false(0) depending whether scalarval is nonzero or zero.

_Bool to scalar

```
scalarval = Boolval ? 1 : 0;
```

scalarval would be 1 or 0 depending on Boolval being true(1) or false(0).

HP-Specific Type Qualifiers

Syntax

```
type-qualifier ::= __thread
```

Description

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

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

Syntax examples:

```
__thread int var;
```

```
int __thread var;
```

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

Only one declaration, for example,

```
__thread int x;
```

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

```
extern __thread int x;
```

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

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

Type Qualifiers

Syntax

```
type-qualifier :: =
   const
   volatile
   __restrict (or restrict for C99)
```

Description

This section describes the *type qualifiers* — `volatile`, `const` and `__restrict` (or `restrict` for C99).

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

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

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

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

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

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

Use the `__restrict` type qualifier on the declaration of a pointer type to indicate that the pointer is subject to compiler optimizations. The `restrict` is a C99 keyword which only supported under C99 mode.

The formal definition of restricted pointer in C99 follows:

1. Let `D` be a declaration of an ordinary identifier that provides a means of designating an object `P` as a restrict-qualified pointer to type `T`.
2. 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 the function that is called at program startup in a freestanding environment).
3. 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`. Note that *based* is defined only for expressions with pointer types.
4. During each execution of `B`, let `L` be any `lvalue` that has `&L` based on `P`. If `L` is used to access the value of the object `X` that it designates, and `X` is also modified (by any means), then the following requirements apply:

   - `T` shall not be const-qualified.
   - Every other lvalue used to access the value of `X` shall also have its address based on `P`.
   - Every access that modifies `X` shall be considered also to modify `P`, for the purposes of this subclause.

   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 these requirements are not met, then the behavior is undefined.
5. Here an execution of `B` means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration associated with `B`.
6. A translator is free to ignore any or all aliasing implications of uses of restrict.

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

Table 3-2 Declarations using const and volatile

Declaration	Meaning
`volatile int vol_int;`	Declares a volatile int variable.

`const int *ptr_to_const_int;`
`int const *ptr_to_const_int;`
	Both declare a variable pointer to a constant int.

Table 3-2 Declarations using const and volatile (Continued)

Declaration	Meaning
`int *const const_ptr_to_int`	Declares a constant pointer to a variable int.
`int *volatile vpi, *pi;`	Declares two pointers: vpi is a volatile pointer to an int; pi is a pointer to an int.
`int const *volatile vpci;`	Declares a volatile pointer to a constant int.
`const *pci;`	Declares a pointer to a constant int. Since no type specifier was given, it defaults to int.

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

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

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

```
volatile int * ptr_to_vol_int;
```

the type would be `pointer to volatile int`.

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

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

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

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

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

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

Structure and Union Specifiers

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

Syntax

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

struct-or-union ::=
     struct
     union

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

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

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

struct-declarator-list ::=
     struct-declarator
     struct-declarator-list   struct-declarator

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

Description

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

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

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

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

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

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

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

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

Refer to Chapter 10, “HP C/HP-UX Implementation Topics,” on page 235 for more information on bit-fields.

Structure and Union Tags

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

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

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

For example, consider the following declarations:

```
struct s1 {
     int x;
     float y;
};

struct s1 obj1, *obj2;
```

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

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

Consider the following example:

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

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

A subsequent declaration may use the defined structure tag:

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

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

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

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

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

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

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

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

Example

```
struct tag1 {
   int m1;
   int   :16;             /* unnamed bit-field */
   int m2 :16;            /* named bit-field; packed into
                          /* same word as previous member */
   int m3, m4;
};                        /* empty declarator list */
union tag2 {
   int u1;
   int    :16;
   int u2:16;             /* bit-field, starts at offset 0 */
   int u3, u4;
} fudge1, fudge2;         /* declarators denoting objects of the union type */

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

Enumeration

The identifiers in an enumeration list are declared as constants.

Syntax

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

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

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

enumeration-constant ::= identifier
```

Description

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

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

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

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

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

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

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

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

Examples

```
enum color {RED, GREEN, BLUE};

enum objectkind {triangle, square=5, circle};  /* circle == 6 */
```

Sized enum - HP C Extension

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

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

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

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

NOTE *`enumeration-constant`*s will have the same size as the type specified in the enumeration declaration.

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

Declarators

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

Syntax

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

direct-declarator  ::=
      identifier
      (declarator)
      direct-declarator [[constant-expression]]
      direct-declarator (parameter-type-list)
      direct-declarator ([identifier-list])
pointer  ::=
      * [type-qualifier-list]
      * [type-qualifier-list] pointer

type-qualifier-list  ::=
      type-qualifier
      type-qualifier-list type-qualifier

parameter-type-list ::=
      parameter-list
      parameter-list , ...

parameter-list ::=
      parameter-declaration
      parameter-list , parameter-declaration

parameter-declaration ::=
      declaration-specifiers declarator
      declaration-specifiers [abstract-declarator]

identifier-list ::=
      identifier
      identifier-list , identifier
```

Description

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

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

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

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

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

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

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

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

Pointer Declarators

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

Some examples:

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

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

Array Declarators

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

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

declares `D` to be an array of type `T`.

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

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

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

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

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

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

Following are examples of array declarations:

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

Note that an array of type `T` that is the formal parameter in a function definition has been converted to a pointer to type `T`. The array name in this case is a modifiable lvalue and can appear as the left operand of an assignment operator. The following function will clear an array of integers to all zeros. Note that the array name, which is a parameter, must be a modifiable lvalue to be the operand of the `++` operator.

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

Variable Length Array

Variable Length Array(VLA) is a part of C99 standards (ISO/IEC 9899:1999). VLA allows the integer expression delimited by `[` and `]` in an array declarator to be a variable expression or *. An identifier whose declaration has such an array declarator is a variably modified (VM) type.

All identifiers having a VM type must be either:

- block scope or function prototype scope, if the expression in an array declarator is a variable expression *or*
- function prototype scope, if the expression is *.

NOTE VLA is supported only in ANSI extended (-Ae) mode.

Arrays declared with the static or extern storage class specifier cannot have a VM type. But a pointer to an array declared with the static storage class specifier can have a VM type. All identifiers having a VM type are ordinary identifiers and therefore cannot be the members of structures or unions.

```
extern int n;

int A[n];                     // Error - file scope VM type
extern int (*p) [n];          // Error - file scope VM type
int B[100];                   // OK - file scope but not VM type
void foo(int m, int C[m]      // OK - function prototype
                              // scope VM type
{
   typedef int VLA[m] [m];    // OK - block scope VM type
   int D[m];                  // OK - block scope with VM type
   static int E[m];           // Error - static specifier in VM type
   extern int F[m];           // Error - extern specifier in VM type
   int (*q)[m];               // OK - block scope with VM type
   extern int (*r)[m];        // Error - extern specifier in VM type
   static int (*s)[m] = &B;   // OK - static specifier allowed in VM
                              // type since 's' is pointer to array
struct tag {
   int (*x)[n];              // Error - x not ordinary identifier
   int y[n];                 // Error - y not ordinary identifier
   };
}
```

A goto statement is not allowed to jump past any declarations of identifiers having a VM type. A jump within the scope is permitted.

```
goto L1;             // Error - going INTO scope of VM type
   {
      int a[n];
      a[j] = 4;
   L1:
      a[j] = 3;
      goto L2;       // OK, going WITHIN scope of VM type
      a[j] = 5;
   L2:
      a[j] = 6;
   }
   goto L2;             // Error - going INTO scope of VM type
```

The size of an object having a VM type is determined and fixed at the point of the object's declaration and cannot be altered.

The following example assumes size of (int) = 4,

```
   int n = 10, vla[n];
   n = 20;
   printf(""%d", sizeof(vla);      // prints 40, not 80
```

Function Declarators

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

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

or

`T D (`*`[identifier-list]`*`)`

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

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

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

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

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

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

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

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

Type Names

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

Syntax

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

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

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

Description

Type names are enclosed in parentheses to indicate a cast operation. The destination type is the type named in the cast; the operand is then converted to that type. A type name is a declaration without the identifier specified. For example, the declaration for an integer is `int i`. If the identifier is omitted, only the integer type `int` remains.

Examples

```
int                int
int *              Pointer to int
int ()             Function returning an int
int *()            Function returning a pointer to int
int (*)()          Pointer to function returning an int
int [3];           Array of 3 int
int *[3];          Array of 3 pointers to int
int (*)[3];         Pointer to an array of 3 int
```

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

Type Definitions Using typedef

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

Syntax

typedef-name ::= identifier

Description

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

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

```
typedef int *pointer;
```

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

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

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

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

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

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

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

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

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

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

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

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

Example

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

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

Initialization

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

Syntax

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

initializer-list ::=
    initializer
    initializer-list , initializer
```

Description

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

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

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

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

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

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

is normally specified as

```
int i = 3;
```

When initializing the members of an aggregate, the initializer is a brace-enclosed list of initializes. In the case of a structure with automatic storage duration, the initializer may be a single expression returning a type compatible with the structure. If the aggregate contains members that are aggregates, this rule applies recursively, with the following exceptions:

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

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

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

produces the following layout:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Some more examples include:

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

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

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

Examples

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

You initialize structures as you do any other aggregate:

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

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

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

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

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

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

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

The union initialization is only available in ANSI mode.

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

Compound Literal

Compound literal provide a mechanism for specifying constants of aggregate or union type. Compound literal is a part of the C99 standards (ISO/IEC 9899:1999: 6.5.2.5). Compound literals are an easy means of initializing an object of type aggregate or union without having to allocate a temporary variable for the object. It is represented as an unnamed object with a type and has an lvalue.

Syntax

```
(type-name) {initializer-list}
```

`type-name` must specify an object type or an array of unknown size. The value of the compound literal is that of an unnamed object initialized by the initializer list. The object has static storage if the compound literal occurs outside the body of the function, otherwise it has automatic storage duration associated with the enclosing blocks.

Examples

The following examples detail the usage of compound literals:

Example 3-1 An Array of Scalars

```
int *p=(int[]) {1,2};
```

In this example, an array of size 2 has been declared, with the first two elements initialized to 1 and 2. `(int [ ]){1,2}` represents the compound literal and it is assigned to a pointer variable `p` of type `int`.

Example 3-2 An Array of Structures

```
struct node
{
   int a;
   int b;
};
struct node *st=(struct node[2]) {1,2,3,4};
```

In this example, a pointer of structures has been initialized with the unnamed compund literal object. `(struct node[2]){1,2,3,4}` is the compound literal and is converted to `struct node *` and assigned to `st`.

Example 3-3 As a Parameter

```
int main()
{
   foo((int [])(1,2,3,4));
}
int foo(int *p)
{
}
```

In this example, a compound literal is passed as a parameter to function `foo()` instead of creating a temporary variable in the function `main()` and then passing the compound literal as a parameter to `foo()`. Compound literals can be passed as parameters to functions eliminating the need of defining a temporary variable in the caller function.

Example 3-4 Element in an Array

```
int *p=(int [10000]){[999]=20};
```

This example shows how a particular `element[999]` in an array of size 10000 can be initialized explicitly.

Example 3-5 An Array of Characters

```
char *c=(char []){"/tmp/testfile"};
```

This example shows how a compound literal is used to initialize an array of characters.

Example 3-6 Constant Compound Literal

```
(const float []) {1e0,1e1};
```

This example shows a constant compound literal.

Example 3-7 Single Compound Literal

```
struct int_list {
                  int car;
                  struct int_list *cdr;
               };
struct int_list endless_zeros = {0. &endless_zeros};
```

This example shows how a single compound literal cannot be used to specify a circularly linked object, since compound literals are unnamed.

Example 3-8 Structure Objects

```
drawline((struct point){1,1},&(struct point){3,3});  /*  call */
drawline(struct point, struct point *) { /* definition */ }
```

This example is for structure objects created using compound literals, which are passed to functions.

Example 3-9 Example 9: Possible Ways of Using Compound Literals

```
int x = (int){5};              // initializing x with 5
int *y = (int *){&x};            // initializing y with address of x
```

The purpose of the above example is only to show the other possible ways of using compound literals.

NOTE Compound literal is recognized only in the C99 mode (`-AC99`).

Function Definitions

A *function definition* introduces a new function.

Syntax

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

Description

A function definition provides the following information about the function:

1. **Type**.

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

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

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

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

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

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

NOTE Function prototypes can be used only in ANSI mode.

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

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

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

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

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

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

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

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

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

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

```
int x[ ];
```

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

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

Examples

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

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

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

This is an example of a function definition without prototypes.

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

Here is an example of a function definition using prototypes.

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

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

inline

HP C supports inlining frequently used functions. These functions can be inlined by specifying them as `inline` either in the function declarator or in the function definition.

NOTE `inline` is available only with `-AC99`. For all other options, you must use `__inline`.

The optimizer uses its own heuristics to inline a function, if necessary.

Examples

The following examples detail the usage of `inline`.

Example 3-10 Using inline in Function Declaration

```
inline void foo(int);     //Function declaration with inline specifier
   main()
   {
      foo(5);
   }
   void foo(int x)
   {
      . . .
   }
```

Example 3-11 Using inline in Function Definition

```
   main()
   {
      foo(5);
   }
   inline foo(int x0)      // Function definition with inline specifier
   {
      . . .
   }
```

Four-Byte Extended UNIX Code (EUC)

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

4 Type Conversions

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

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

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

Integral Promotions

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

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

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

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

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

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

ANSI C gives the following results:

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

To avoid situations where unsigned preserving and value preserving promotion rules yield different results, you could refrain from using an unsigned char or unsigned short in an expression that is used as an operand of one of the following operators: `>>`, `/`, `%`, `<`, `<=`, `>`, or `>=`. Or remove the ambiguity by using an explicit cast to specify the conversion you want.

If you enable ANSI migration warnings, the compiler will warn you of situations where differences in the promotion rules might cause different results. See “Compiling and Running HP C Programs” on page 207 for information on enabling ANSI migration warnings.

Usual Arithmetic Conversions

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

1. If either operand is long double, the other operand is converted to `long double`.
2. If either operand is double, the other operand is converted to `double`.
3. If either operand is float, the other operand is converted to `float`.
4. Integral promotions are performed on both operands, and then the rules listed below are followed. These rules are a strict extension of the ANSI "Usual Arithmetic Conversions" rule (Section 3.2.1.5). This extension ensures that integral expressions will involve `long long` only if one of the operands is of type `long long`. For ANSI conforming compilation, the integral promotion rule is as defined in Section 3.2.1.1 of the Standard. For non-ANSI compilation, the unsigned preserving promotion rule is used.

   a. If either operand is `unsigned long long`, the other operand is converted to `unsigned long long`,

   b. otherwise, if one operand is `long long`, the other operand is converted to `long long`,

   c. otherwise, if either operand is `unsigned long int`, the other operand is converted to `unsigned long int`,

   d. otherwise, if one operand is `long int`, and the other is `unsigned int`, and `long int` can represent all the values of an `unsigned int`, then the `unsigned` int is converted to a `long int`. (If one operand is `long int`, and the other is `unsigned int`, and `long int` can NOT represent all the values of an `unsigned int`, then both operands are converted to `unsigned long int`.)

   e. If either operand is `long int`, the other operand is converted to `long int`.

   f. If either operand is `unsigned int`, the other operand is converted to unsigned int.

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

NOTE In compatibility mode, the rules are slightly different.

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

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

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

Arithmetic Conversions

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

Integral Conversions

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

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

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

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

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

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

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

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

Floating Conversions

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

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

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

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

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

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

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

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

Arrays, Pointers, and Functions

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

In most cases, when an expression with array type is used, it is automatically converted to a pointer to the first element of the array. As a result, array names and pointers are often used interchangeably in C. This automatic conversion is not performed in the following contexts:

- When the array is the operand of `sizeof` or the unary `&`;.
- It is a character string literal initializing an array of characters.
- It is a wide string literal initializing an array of wide characters.

5 Expressions and Operators

Operators are grouped as follows:

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

See Also:

- Operator Precedence
- Operator Quick Reference

The different types of **expressions** are:

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

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

Syntax

exp1 + *exp2*	Adds *exp1* and *exp2*. An *exp* can be any integer or floating-point expression.
exp1 - *exp2*	Subtracts *exp2* from *exp1*.
exp1 * *exp2*	Multiplies *exp1* by *exp2*.
exp1 / *exp2*	Divides *exp1* by *exp2*.
exp1 % *exp2*	Finds modulo of *exp1* divided by *exp2*. (That is, finds the remainder of an integer division.) An expression can be any integer expression.
-*exp*	Negates the value of *exp*.
+*exp*	Identity (unary plus).

Arguments

exp	Any constant or variable expression.

Description

The addition, subtraction, and multiplication (`+`, `-`, and `*`) operators perform the usual arithmetic operations in C programs. The operands may be any integral or floating-point value, with the following exception:

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

The addition and subtraction operators also accept pointer types as operands. Pointer arithmetic is described in “Pointer Operators (*, ->, &)” on page 118.

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

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

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

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

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

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

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

Refer to “Operator Precedence” on page 135 for information about how these and other operators evaluate with respect to each other.

Array Subscripting ([ ])

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

Syntax

```
 postfix-expression [ expression ]
```

Description

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

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

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

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

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

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

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

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

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

As another example, the expression

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

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

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

Syntax

lvalue = *expression*	Simple assignment.
lvalue += *expression*	Addition and assignment.
lvalue -= *expression*	Subtraction and assignment.
lvalue *= *expression*	Multiplication and assignment.
lvalue /= *expression*	Division and assignment.
lvalue %= *expression*	Modulo division and assignment.
lvalue <<= *expression*	Left shift and assignment.
lvalue >>= *expression*	Right shift and assignment.
lvalue &= *expression*	Bitwise AND and assignment.
lvalue ^= *expression*	Bitwise XOR and assignment.
lvalue |= *expression*	Bitwise OR and assignment.

Arguments

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

Description

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

The Assignment Operator (=)

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

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

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

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

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

is interpreted as

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

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

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

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

```
j = f = 3.5;
```

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

The Other Assignment Operators

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

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

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

```
i += 10;
```

In other words, any statement of the form

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

can be represented in the following shorthand form:

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

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

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

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

The second statement is ambiguous because C does not specify which assignment operand is evaluated first. See “Operator Precedence” on page 135 for more information concerning order of evaluation.

Assignment Type Conversions

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

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

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

```
c = 882;
```

The binary representation of 882 is

```
00000011 01110010
```

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

```
01110010
```

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

```
x / (U_MAX+1)
```

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

```
j = 71124;
```

assigns to `j`, the remainder of

```
71124 / (65535+1)
```

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

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

```
f = 10;
```

is executed as if it had been written

```
f = 10.0;
```

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

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

If you compile and execute this program, you get:

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

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

```
j = 2.5;
```

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

```
j = -5.8;
```

the value of `j` is -5.

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

```
j = 999999999999.0
```

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

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

```
f = 1.0123456789
```

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

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

The output is

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

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

```
f = 2e40;
```

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

Example

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

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

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

Syntax

exp1 << *exp2*	Left shifts (logical shift) the bits in *exp1* by *exp2* positions.
exp1 >> *exp2*	Right shifts (logical or arithmetic shift) the bits in *exp1* by *exp2* positions.
exp1 & *exp2*	Performs a bitwise AND operation.
exp1 ^ *exp2*	Performs a bitwise OR operation.
exp1 | *exp2*	Performs a bitwise inclusive OR operation.
~*exp1*	Performs a bitwise negation (one's complement) operation.

Arguments

exp1 and *exp2*	Any integer expression.

Description

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

Bit-Shift Operators

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

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

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

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

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

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

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

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

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

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

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

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

```
10 >> 50
```

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

Bit Logical Operators

The logical bitwise operators are similar to the Boolean operators, except that they operate on every bit in the operand(s). For instance, the bitwise AND operator (&) compares each bit of the left operand to the corresponding bit in the righthand operand. If both bits are 1, a 1 is placed at that bit position in the result. Otherwise, a 0 is placed at that bit position.

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

Table 5-1 Truth Table for the bitwise AND operator, (&)

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

Table 5-1 Truth Table for the bitwise AND operator, (&) (Continued)

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

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

Table 5-2 The Bitwise AND Operator

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

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

Table 5-3 Truth Table

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

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

Table 5-4 Example Using the Bitwise Inclusive OR Operator

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

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

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

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

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

Table 5-6 Example Using the XOR Operator

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

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

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

bit x of op2	result
0	0
0	1

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

Table 5-8 Example Using the Bitwise Complement Operator

Expression	Hexadecimal Value	Binary Representation
9430	0x24d6	00100100 11010110

Table 5-8 Example Using the Bitwise Complement Operator (Continued)

Expression	Hexadecimal Value	Binary Representation
~9430	0xdb29	11011011 00101001

Cast Operator

Syntax

(*data_type*) *exp*

Arguments

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

Description

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

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

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

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

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

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

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

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

A cast expression may not be an lvalue.

Casting Integers to Other Integers

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

Table 5-9 Integer Conversions

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

* Case C for `long` in 64-bit mode.

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

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

```
                     hex    dec
char i =              37     55
(short) i  =>       0037     55
(int) i    => 00000037     55

char j =              c3    -61
(short) j  =>       ffc3    -61
(int) j    => ffffffc3    -61
```

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

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

```
                          hex          dec
signed long int   i = cf34bf1    217271281

(signed short int)i  =>  4bf1        19441
(signed char)i       =>    f1          -15
(unsigned char)i     =>    f1           241
```

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

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

```
                          hex          dec          hex     dec
signed int  i     =    fffffca9         -855     0000f2a1   62113

(unsigned int)i  =>    fffffca9   4294966441     0000f2a1   62113
```

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

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

```
                             hex          dec
signed short int   i =      ff55         -171

(unsigned long int)i =>   fffff55   4294967125
```

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

Here are some more examples:

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

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

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

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

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

Casting Enumerated Values to Integers

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

Casting Double to Float and Vice Versa

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

Casting Pointers to Pointers

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

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

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

Comma Operator (,)

Syntax

exp1, *exp2*

Arguments

exp1	Any expression.
exp2	Any expression.

Description

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

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

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

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

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

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

Conditional Expression Operator (`?:`)

Syntax

exp1 ? *exp2* : *exp3*

Arguments

exp1	Any expression.
exp2	Any expression.
exp3	Any expression.

Description

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

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

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

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

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

Example

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

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

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

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

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

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

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

Function Calls

Syntax

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

Description

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

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

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

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

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

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

Function Arguments

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

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

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

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

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

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

Function Formal Parameters

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

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

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

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

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

Notice that the addresses of the objects (pointers to `int`) were passed and not the objects themselves.

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

Function Recursion

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

Increment and Decrement Operators (++, --)

Syntax

`lvalue``++`	Increments the current value of *`lvalue`* after *`lvalue`* has been referenced.
`lvalue``--`	Decrements the current value of *`lvalue`* after *`lvalue`* has been referenced.
`++`*`lvalue`*	Increments the current value of *`lvalue`* before *`lvalue`* is referenced.
`--`*`lvalue`*	Decrements the current value of *`lvalue`* before *`lvalue`* has been referenced.

Arguments

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

Description

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

The increment and decrement operators are unary. The operand must be a scalar lvalue — it is illegal to increment or decrement a constant, structure, or union. It is legal to increment or decrement pointer variables, but the meaning of adding 1 to a pointer is different from adding 1 to an arithmetic value. This is described in “Pointer Operators (*, ->, &)” on page 118.

Postfix and Prefix Forms

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

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

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

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

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

```
x++;
```

is equivalent to

```
++x;
```

Similarly, the statement

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

is equivalent to

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

Using Increment and Decrement within Expressions Consider the following function that inserts newlines into a text string at regular intervals.

```
#include <stdio.h>

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

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

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

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

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

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

If the right operand is evaluated first, the expression becomes

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

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

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

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

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

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

Precedence of Increment and Decrement Operators

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

```
--j++
```

is evaluated as

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

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

Examples

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

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

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

The result is as follows:

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

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

Logical Operators (`&&`, `||`, `!`)

Syntax

exp1 && *exp2*	Logical AND.
exp1 || *exp2*	Logical OR.
`!`*exp1*	Logical NOT.

Arguments

exp1	Any expression.
exp2	Any expression.

Description

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

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

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

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

```
1 && -5
```

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

```
0.5 && -5
```

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

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

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

Truth Table for C's Logical Operators

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

Table 5-10 Truth Table for C's Logical Operators

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

Examples of Expressions Using the Logical Operators

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

Table 5-11 Examples of Expressions Using the Logical Operators

Given the following declarations: `int j = 0, m = 1, n = -1;` `float x = 2.5, y = 0.0;`		
Expression	**Equivalent Expression**	**Result**
`j && m`	`(j) && (m)`	0

Table 5-11 Examples of Expressions Using the Logical Operators

Given the following declarations:
`int j = 0, m = 1, n = -1;`
`float x = 2.5, y = 0.0;`		

Expression	**Equivalent Expression**	**Result**
`j < m && n < m`	`(j < m) && (n < m)`	1
`m + n || ! j`	`(m + n) || (!j)`	1
`x * 5 && 5 || m / n`	`((x * 5) && 5) || (m / n)`	1
`j <= 10 && x >= 1 && m`	`((j <= 10) && (x >= 1)) && m`	1
`!x || !n || m+n`	`((!x) || (!n)) || (m+n)`	0
`x * y < j + m || n`	`((x * y) < (j + m)) || n`	1
`(x > y) + !j || n++`	`((x > y) + (!j)) || (n++)`	1
`(j || m) + (x || ++n)`	`(j || m) + (x || (++n))`	2

Side Effects in Logical Expressions

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

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

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

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

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

Example

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

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

    won_lottery = enough_vacation = money_saved = 0;

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

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

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

    printf("\n");
    if (won_lottery)
    {
        printf("Why do you need a program to decide if you");
        printf(" can afford a trip to Europe?\n");
    }  /*  end if  */
    if (won_lottery || (enough_vacation &&money_saved))
```

```
        printf("Look out Paris!\n");
    else if (enough_vacation &&(!money_saved))
        printf("You've got the time, but you haven't got \
the dollars.\n");
    else if (!enough_vacation || (!money_saved))
    {
        printf("Tough luck. Try saving your money and ");
        printf("vacation days next year.\n");
    }  /*  end else/if  */
}
```

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

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

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

Pointer Operators (*, ->, &)

Syntax

******ptr_exp*	Dereferences a pointer. That is, finds the contents stored at the virtual address that *ptr_exp* holds.
ptr->member	Dereferences a *ptr* to a structure or union where *member* is a member of that structure or union.
&*lvalue*	Finds the virtual address where the lvalue stored.

Description

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

Assigning an Address Value to a Pointer

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

```
long *ptr;
```

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

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

But this is illegal:

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

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

```
/* Program name is "ptr_example1". */
#include <stdio.h>
int main()
```

```
{
    int j = 1;
    int *pj;
    pj = &j;  /* Assign the address of j to pj */
    printf("The value of j is: %d\n", j);
    printf("The address of j is: %p\n", pj);
}
```

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

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

Dereferencing a Pointer

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

```
/* Program name is "ptr_example2". */

#include <stdio.h>

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

The output from this program looks something like this:

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

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

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

```
float *fp;
```

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

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

```
*fp = 3.15;
```

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

```
fp = 3.15;
```

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

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

```
/* Program name is "ptr_example3". */

#include <stdio.h>

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

The result is

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

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

```
cc: "ptr_example3.c", line 9: warning 604: Pointers are not
    assignment-compatible.
```

Pointer Arithmetic

The following arithmetic operations with pointers are legal:

- You may add an integer to a pointer or subtract an integer from a pointer.
- You may use a pointer as an operand to the ++ and operators.
- You may subtract one pointer from another pointer, if they point to objects of the same type.
- You may compare two pointers

All other arithmetic operations with pointers are illegal.

When you add or subtract an integer to or from a pointer, the compiler automatically scales the integer to the pointer's type. In this way, the integer always represents the number of objects to jump, not the number of bytes. For example, consider the following program fragment:

```
int x[10], *p1x = x, *p2x;

p2x = p1x + 3;
```

Since pointer `p1x` points to a variable (`x`) that is 4 bytes long, then the expression `p1x + 3` actually increments `p1x` by `12 (4 * 3)`, rather than by `3`.

It is legal to subtract one pointer value from another, provided that the pointers point to the same type of object. This operation yields an integer value that represents the number of objects between the two pointers. If the first pointer represents a lower address than the second pointer, the result is negative. For example,

```
&a[3] - &a[0]
```

evaluates to 3, but

```
&a[0] - &a[3]
```

evaluates to -3.

It is also legal to subtract an integral value from a pointer value. This type of expression yields a pointer value. The following examples illustrate some legal and illegal pointer expressions:

```
long *p1, *p2;
int a[5], j;
char *p3;
```

```
p1 = a;       /* Same as p1 = &a[0] */
p2 = p1 + 4;  /* legal */
j = p2 - p1;  /* legal -- j is assigned 4  */
j = p1 - p2;  /* legal -- j is assigned -4 */
p1 = p2 - 2;  /* legal -- p2 points to a[2] */
p3 = p1 - 1;  /* ILLEGAL -- different pointer types*/
j = p1 - p3;  /* ILLEGAL -- different pointer types*/
j = p1 + p2;  /* ILLEGAL -- cannot add pointers */
```

Arrays and Pointers

Arrays and pointers have a close relationship in the C language. You can exploit this relationship in order to write more efficient code. See the discussion of "Array Subscripting ([ ])" on page 86 for more information.

Casting a Pointer's Type

A pointer to one type may be cast to a pointer to any other type. For example, in the following statements, a pointer to an `int` is cast to a pointer to a `char`. Presumably, the function `func()` expects a pointer to a `char`, not a pointer to an `int`.

```
int i, *p = &i;
func((char *) p);
```

As a second example, a pointer to a `char` is cast to a pointer to `struct H`:

```
struct H {
    int q;
} x, y;
char  *genp = &x;
y = (struct H *)genp->q;
```

See "Cast Operator" on page 98 for more information about the cast operator.

It is always legal to assign any pointer type to a generic pointer, and vice versa, without a cast. For example:

```
float x, *fp = &x;
int j, *pj = &j;
void *pv;
pv = fp;  /* legal */
fp = pv;  /* legal */
```

In both these cases, the pointers are implicitly cast to the target type before being assigned.

Null Pointers

The C language supports the notion of a **null pointer** — that is, a pointer that is guaranteed not to point to a valid object. A null pointer is any pointer assigned the integral value 0. For example:

```
char *p;
p = 0;  /* make p a null pointer */
```

In this one case — assignment of 0 — you do not need to cast the integral expression to the pointer type.

Null pointers are particularly useful in control-flow statements, since the zero-valued pointer evaluates to false, whereas all other pointer values evaluate to true. For example, the following `while` loop continues iterating until `p` is a null pointer:

```
char *p;
 . . .
while (p) {
 . . .
/* iterate until p is a null pointer */
 . . .
}
```

This use of null pointers is particularly prevalent in applications that use arrays of pointers.

The compiler does not prevent you from attempting to dereference a null pointer; however, doing so may trigger a run-time access violation. Therefore, if it is possible that a pointer variable is a null pointer, you should make some sort of test like the following when dereferencing it:

```
if (px && *px)   /* if px = 0, expression will short-circuit
  . . .             before dereferencing occurs*/
```

Null pointers are a portable feature.

Example 5-1 Accessing a 1-Dimensional Array Through Pointers

```
/* Program name is "pointer_array_example1". This program
 * shows how to access a 1-dimensional array through
 * pointers. Function count_chars returns the number of
 * characters in the string passed to it.
 * Note that *arg is equivalent to a_word[0];
 * arg + 1 is equivalent to a_word[1]...
 */
#include <stdio.h>

int count_chars(char *arg)
```

```
{
    int count = 0;
    while (*arg++)
        count++;
    return count;
}

int main(void)
{
    char a_word[30];
    int number_of_characters;
    printf("Enter a word -- ");
    scanf("%s", a_word);
    number_of_characters = count_chars(a_word);
    printf("%s contains %d characters.\n", a_word,
            number_of_characters);
}
```

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

```
Enter a word -- Marilyn
Marilyn contains 7 characters.
```

Example 5-2 Accessing a 2-Dimensional Array Through Pointers

```
/*  Program name is "pointer_array_example2".  This program
 *  demonstrates two ways to access a 2-dimensional array. */
#include <stdio.h>

int main(void)
{
    int count = 0, which_name;
    char c1, c2;
    static char str[5][10] = {"Phil", "Sandi", "Barry",
                              "David", "Amy"};
    static char *pstr[5] = { str[0], str[1], str[2],
                             str[3], str[4]};
/*  pstr is an array of pointers.  Each element in the array
 *  points to the beginning of one of the arrays in str.
 */
/* Prompt for information. */
    printf("Which name do you want to retrieve?\n");
    printf("Enter 0 for the first name,\n");
    printf("      1 for the second name, etc. -- ");
    scanf("%d", &which_name);
/* Print name directly through array. */
    while (c1 = str[which_name][count++])
```

```
        printf("%c", c1);
    printf("\n");

/* Print same name indirectly through an array of pointers. */
    while (c2 = *(pstr[which_name]++))
        printf("%c", c2);
/* We could also have used the following statement instead of
 * the two previous ones:  printf("%s", pstr[which_name]);
 */
    printf("\n");
}
```

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

```
Which name do you want to retrieve?
Enter 0 for the first name,
      1 for the second name, etc. -- 1
Sandi
Sandi
```

Relational Operators (>, >=, <, ==, !=)

Syntax

exp1 > *exp2*	Greater than.
exp1 >= *exp2*	Greater than or equal to.
exp1 < *exp2*	Less than.
exp1 <= *exp2*	Less than or equal to.
exp1 == *exp2*	Equal to.
exp1 != *exp2*	Not equal to.

Arguments

exp1	Any expression.
exp2	Any expression.

Description

A relational expression consists of two expressions separated by one of six relational operators. The relational expression evaluates either to 1 (true) or 0 (false).

The equality operator (==) performs the same function as Pascal's = or Fortran's `.EQ.`; it just looks different. Although the equality operator looks similar to the assignment operator (=), the two operators serve completely different purposes. Use the assignment operator when you want to assign a value to a variable, but use the equality operator when you want to test the value of an expression.

Confusing = with ==

One of the most common mistakes made by beginners and experts alike is to use the assignment operator (=) instead of the equality operator (==). For instance:

```
while (j = 5)
   do_something();
```

What is intended, clearly, is that the `do_something()` function should only be invoked if `j` equals five. It should be written

```
while (j == 5)
  do_something();
```

The first version is syntactically legal, since all expressions have a value. The value of the expression `j = 5` is `5`. Since this is a nonzero value, the while expression will always evaluate to true and `do_something()` will always be invoked.

Relational Operators Precedence Rules

Relational operators have lower precedence than arithmetic operators. The expression

```
a + b * c < d / f
```

is evaluated as if it had been written

```
(a + (b * c)) < (d / f)
```

Among the relational operators, `>, >=, <, and <=` have the same precedence. The `==` and `!=` operators have lower precedence. All of the relational operators have left-to-right associativity. The following table illustrates how the compiler parses complex relational expressions.

Table 5-12 Examples of Expressions Using the Relational Operators

Given the following declaration:
`int j = 0, m = 1, n = -1;`
`float x = 2.5, y = 0.0;`

Expression	Equivalent Expressions	Result
`j > m`	`j > m`	0
`m / n < x`	`(m / n) < x`	1
`j <= m >= n`	`((j <=m) >= n)`	1
`j <= x == m`	`((j <= x) == m)`	1
`- x + j == y > n > m`	`((-x) + j) == ((y > n) >= m)`	0
`x += (y >= n)`	`x = (x + (y >= n))`	3.5
`++j == m != y * 2`	`((++j) == m) != (y * 2)`	1

Evaluation of Relational Expressions

Relational expressions are often called Boolean expressions, in recognition of the nineteenth-century mathematician and logician, George Boole. Many programming languages, such as Pascal, have Boolean data types for representing true and false. The C language, however, represents these values with integers. Zero is equivalent to false, and any nonzero value is considered true.

The value of a relational expression is an integer, either 1 (indicating the expression is true) or 0 (indicating the expression is false). The examples in the following table illustrate how relational expressions are evaluated:

Table 5-13 Relational Expressions

Expression	Value
`-1 < 0`	`1`
`0 > 1`	`0`
`5 == 5`	`1`
`7 != -3`	`1`
`1 >= -1`	`1`
`1 > 10`	`0`

Because Boolean values are represented as integers, you can write

```
if (j)
   statement;
```

If `j` is any nonzero value, *`statement`* is executed; if `j` equals 0, *`statement`* is skipped. Likewise, the statement

```
if (isalpha(ch))
```

is exactly the same as

```
if (isalpha(ch) != 0)
```

The practice of using a function call as a Boolean expression is a common idiom in C. It is especially effective for functions that return 0 if an error occurs, since you can use a construct such as

```
if (func())
    proceed;
else
    error handler;
```

Dangers of Comparing Floating-Point Values

You may get unexpected results if you compare floating-point values for equality because floating-point representations are inexact for some numbers. For example, the following expression, though algebraically true, will evaluate to false on many computers:

```
(1.0/3.0 + 1.0/3.0 + 1.0/3.0) == 1.0
```

This evaluates to 0 (false) because the fraction 1.0/3.0 contains an infinite number of decimal places (3.33333…). The computer is only capable of holding a limited number of decimal places, so it rounds each occurrence of 1/3. As a result, the left side of the expression does not equal exactly 1.0.

This problem can occur in even more subtle ways. Consider the following code:

```
double divide(double num, double denom)
{
    return num/denom;
}
int main(void)
{
    double c, a = 1.0, b = 3.0;
    c = a/b;
    if (c != divide(a, b))
        printf("Fuzzy doubles\n");
}
```

Surprisingly, the value stored in `c` may not equal the value returned by `divide()`. This anomaly occurs due to the fact that some computers can represent more decimal places for values stored in registers than for values stored in memory. Because the value returned by divide() is never stored in memory, it may not be equal to the value `c`, which has been rounded for memory storage.

NOTE To avoid bugs caused by inexact floating-point representations, you should refrain from using strict equality comparisons with floating-point types.

Example 5-3 C's Relational Operators in Action

```
/*  Program name is "relational_example". This program
 *  does some mathematical calculations and shows
 *  C's relational operators in action.
 */
#include <stdio.h>

int main(void)
{
    int num, i;
    printf("\n");
    num = 5;
    printf("The number is: %d\n", num);
    for (i = 0; i <= 2; i++)
    {
        if (num < 25)
        {
            num *= num;
            printf("The number squared is: %d\n", num);
        }
        else if (num == 25) {
            num *= 2;
            printf("Then, when you double that, you get: %d\n", num);
        }
        else if (num > 25)
        {
            num -= 45;
            printf("And when you subtract 45, you're back where ");
            printf("you started at: %d\n", num);
        } /* end if */
    } /* end for */

    if (num != 5)
        printf("The programmer made an error in setting up this \
example\n");
}
```

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

```
The number is: 5
The number squared is: 25
Then, when you double that, you get: 50
And when you subtract 45, you're back where you started at: 5
```

sizeof Operator

Syntax

`sizeof` *`exp`*;

`sizeof (`*`type_name`*`)`

Arguments

exp	An expression of any type except function, void, or bit field.
type_name	The name of a predefined or user-defined data type, or the name of some variable. An example of a predefined data type is `int`. A user-defined data type could be the tag name of a structure.

Description

The `sizeof` unary operator finds the size of an object. It accepts two types of operands: an expression or a data type. If the type of the operand is a variable length array, the operand is evaluated - the compiler only determines what type the result would be for the expression operand. Any side effects in the expression, therefore, will not have an effect. The result type of the `sizeof` operator is `size_t`.

If the operand is an expression, `sizeof` returns the number of bytes that the result occupies in memory:

```
/* Returns the size of an int (4 if ints are four bytes long) */
sizeof(3 + 5)

/* Returns the size of a double (8 if doubles are
 *  eight bytes long)
 */
sizeof(3.0 + 5)

/*  Returns the size of a float (4 if floats are
 *  four bytes long)
 */
float x;
sizeof(x)
```

For expressions, the parentheses are optional, so the following is legal:

```
sizeof x
```

By convention, however, the parentheses are usually included.

The operand can also be a data type, in which case the result is the length in bytes of objects of that type:

```
sizeof(char)     /* 1 on all machines */
sizeof(short)    /* 2 on HP 9000 Series */
sizeof(float)    /* 4 on HP 9000 Series  */
sizeof(int *)    /* 4 on HP 9000 Series */
```

The parentheses are required if the operand is a data type.

NOTE The results of most `sizeof` expressions are implementation dependent. The only result that is guaranteed is the size of a `char`, which is always 1.

In general, the `sizeof` operator is used to find the size of aggregate data objects such as arrays and structures.

Example

You can use the `sizeof` operator to obtain information about the sizes of objects in your C environment. The following prints the sizes of the basic data types:

```
/*  Program name is "sizeof_example".  This program
 *  demonstrates a few uses of the sizeof operator.
 */
#include <stdio.h>
int main(void)
{
    printf("TYPE\t\tSIZE\n\n");
    printf("char\t\t%d\n", sizeof(char));
    printf("short\t\t%d\n", sizeof(short));
    printf("int\t\t%d\n", sizeof(int));
    printf("float\t\t%d\n", sizeof(float));
    printf("double\t\t%d\n", sizeof(double));
}
```

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

```
TYPE         SIZE

char         1
short        2
int          4
float        4
double       8
```

Structure and Union Members (., ->)

A member of a structure or a union can be referenced using either of two operators: the period or the right arrow.

Syntax

```
 postfix-expression . identifier postfix-expression -> identifier
```

Description

Use the period to reference members of structures and unions directly. Use the arrow operator to reference members of structures and unions pointed to by pointers. The arrow operator combines the functions of indirection through a pointer and member selection. If `P` is a pointer to a structure with a member `M`, the expression `P->M` is identical to `(*P).M`.

The *postfix-expression* in the first alternative must be a structure or a union. The expression is followed by a period (.) and an identifier. The identifier must name a member defined as part of the structure or union referenced in the *postfix-expression*. The value of the expression is the value of the named member. It is an lvalue if the *postfix-expression* is an lvalue.

If the *postfix-expression* is a pointer to a structure or a pointer to a union, follow it with an arrow (composed of the `-` character followed by the `|`) and an identifier. The identifier must name a member of the structure or union which the pointer references. The value of the primary expression is the value of the named member. The resulting expression is an lvalue.

The `.` operator and the `->` operator are closely related. If `S` is a structure, `M` is a member of structure `S`, and `&S` is a valid pointer expression, `S.M` is the same as `(&S)->M`.

Operator Precedence

Precedence is the order in which the compiler groups operands with operators. The C compiler evaluates certain operators and their operands before others. If operands are not grouped using parentheses, the compiler groups them according to its own rules.

The following lists C operator precedence in `highest` to `lowest` precedence:

Table 5-14 C Operator Precedence

Class of operator	Operators	Grouping
primary	`()` `[]` `->` `.`	left to right
unary	(type casting)	right to left
	`sizeof`	
	`&` (address of)	
	`*` (dereference)	
	`-` (reverse sign)	
	`~` `!`	
	`++` `--`	
multiplicative	`*` `/` `%`	left to right
additive	`+` `-`	left to right
shift	`<<` `>>`	left to right
relational	`<` `<=` `>` `>=`	left to right
equality	`==` `!=`	left to right
bitwise AND	`&`	left to right
bitwise XOR	`^`	left to right
bitwise OR	`|`	left to right
logical AND	`&&`	left to right
logical OR	`||`	left to right

Table 5-14 C Operator Precedence (Continued)

Class of operator	Operators	Grouping
conditional	`?:`	right to left
assignment	`=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`	right to left
comma	`,`	left to right

Precedence among Operators of Same Class

Most operators group from the left to the right but some group from the right to the left. The grouping indicates how an expression containing several operators of the same precedence will be evaluated. Left to right grouping means the expression

```
a/b * c/d
```

behaves as if it had been written:

```
(((a/b)*c)/d)
```

Likewise, an operator that groups from the right to the left causes the expression

```
a = b = c
```

to behave as if it had been written:

```
a = (b = c)
```

Operator Quick Reference

Table 5-15 C Operators

Symbol	Meaning
!	logical negation
!=	inequality
%	remainder
&	AND (bitwise) and address-of
&&	AND (logical)
()	cast and function call
*	multiplication and indirection
+	addition and unary plus
++	increment, prefix or postfix
,	comma
-	subtraction and unary minus
--	decrement, prefix or postfix
->	structure/union pointer (arrow)
.	structure/union member (dot)
/	division
<	less-than
<<	left-shift
<=	less-than-or-equal-to
=	assignment
==	equality
>	greater-than

Table 5-15 C Operators (Continued)

Symbol	Meaning
>=	greater-than-or-equal-to
>>	right shift
?:	conditional
[ ]	subscript
^	OR (bitwise exclusive)
|	OR (bitwise inclusive)
| |	OR (logical)
~	complement
op=	assignment, compound
sizeof	compute object size at translation-time

Constant Expressions

Constant expressions contain only constant values. For example, the following are all constant expressions:

- `5`
- `5 + 6 * 13 / 3.0`
- `'a'`

Integral Expressions

Integer expressions are expressions that, after all automatic and explicit type conversions, produce a result that has one of the integer types. If `j` and `k` are integers, the following are all integral expressions:

- `j`
- `j * k`
- `j / k + 3`
- `k - ‘a’`
- `3 + (int) 5.0`

Floating-Point Expressions

Floating-point expressions are expressions that, after all automatic and explicit type conversions, produce a result that has one of the floating-point types. If `x` is a `float` or `double`, the following are floating-point expressions:

- `x`
- `x + 3`
- `x / y * 5`
- `3.0`
- `3.0 - 2`
- `3 + (float) 4`

lvalue Expressions

An **lvalue** (pronounced “el-value”) is an expression that refers to a region of storage that can be manipulated.

For example, all simple variables, like `int`s and `float`s are lvalues. An element of an array is also an lvalue; however an entire array is not. A member of a structure or union is an lvalue; an entire structure or union is not.

Given the following declarations:

```
int *p, a, b;
int arr[4];
int func();

   a            /* lvalue */
   a + b        /* Not an lvalue */
   p            /* lvalue */
   *p           /* lvalue */
   arr          /* lvalue, but not modifiable */
   *(arr + a)   /* lvalue */
   arr[a]       /* lvalue, equivalent to *(arr+a) */
   func         /* Not an lvalue */
   func()       /* Not an lvalue */
```

Pointer Expressions

Pointer expressions are expressions that evaluate to an address value. These include expressions containing pointer variables, the address-of operator (`&`), string literals, and array names. If `p` is a pointer and `j` is an `int`, the following are pointer expressions:

```
p
&j
p + 1
"abc"
(char *) 0x000fffff
```

Evaluation of Expressions

Expressions are evaluated at run time. The results of the evaluation are called by product values. For many expressions, you won't know or care what this byproduct is. In some expressions, though, you can exploit this feature to write more compact code.

Examples

The following expression is an assignment.

```
x = 6;
```

The value 6 is both the byproduct value and the value that gets assigned to `x`. The byproduct value is not used.

The following example uses the byproduct value:

```
y = x = 6;
```

The equals operator binds from right to left; therefore, C first evaluates the expression `x = 6`. The byproduct of this operation is `6`, so C sees the second operation as

```
y = 6
```

Now, consider the following relational operator expression:

```
(10 < j < 20)
```

It is incorrect to use an expression like this to find out whether `j` is between 10 and 20. Since the relational operators bind from left to right, C first evaluates

```
10 < j
```

The byproduct of a relational operation is 0 if the comparison is false and 1 if the comparison is true.

Assuming that `j` equals `5`, the expression `10 < j` is false. The byproduct will be 0. Thus, the next expression evaluated:

```
0 < 20
```

is true (or 1). This is not the expected answer when `j` equals `5`.

Finally, consider the following fragment:

```
static char  a_char, c[20] = {"Valerie"}, *pc = c;

  while (a_char = *pc++) {
             .   .   .
```

This `while` statement uses C's ability to both assign and test a value. Every iteration of `while` assigns a new value to variable `a_char`. The byproduct of an assignment is equal to the value that gets assigned. The byproduct value will remain nonzero until the end of the string is reached. When that happens, the byproduct value will become 0 (false), and the `while` loop will end.

Evaluation Order of Subexpressions

The C language does not define the evaluation order of subexpressions within a larger expression except in the special cases of the `&&`, `||`, `?:`, and `,` operators. When programming in other computer languages, this may not be a concern. C's rich operator set, however, introduces operations that produce side effects. The `++` operator is a prime example. The `++` operator increments a value by 1 and provides the value for further calculations. For this reason, expressions such as

```
b = ++a*2 + ++a*4;
```

are dangerous. The language does not specify whether the variable `a` is first incremented and multiplied by 4 or is first incremented and multiplied by 2. The value of this expression is undefined.

6 Statements

A program consists of a number of statements that are executed in sequence. A statement can be one of the following types:

- Assignment, where values, usually the results of calculations, are stored in variables.
- Input/Output, data is read in or printed out.
- Control, the program makes a decision about what to do next.

All statements end with a semicolon. Statements are grouped as follows:

- Branch Statements
- Compound Statement or Block
- Expression and Null Statements
- Labeled Statements
- Looping Statements
- Selection Statements

The following is a list of HP C statements:

- break
- continue
- do…while
- for
- goto
- if
- return
- switch
- while

Branch Statements

Syntax

```
goto label;
goto *expression;
continue;
break;
return [expression];
```

Description

Branch statements transfer control unconditionally to another place in the executing program. The branch statements are `goto`, `continue`, `break`, and `return`.

Examples

These four fragments all accomplish the same thing (they print out the multiples of 5 between 1 and 100):

```
 i = 0;
   while (i < 100)
   {
      if (++i % 5)
        continue; /* unconditional jump to top of while loop */
      printf ("%2d ", i);
   }
   printf ("\n");

   i = 0;
L: while (i < 100)
   {
      if (++i % 5)
         goto L:  /* unconditional jump to top of while loop */
      printf ("%2d ",i);
   }
   printf ("\n");


   i = 0;
   while (1)
   {
      if ((++i % 5) == 0)
         printf ("%2d ", i);
      if (i > 100)
         break;   /* unconditional jump past the while loop  */
   }
   printf ("\n");

 i = 0;
   while (1)
   {
      if ((++i % 5) == 0)
         printf ("%2d ", i);
      if (i > 100) {
         printf ("\n");
         return;    /* unconditional jump to calling function  */
      }
   }
```

Compound Statement or Block

Syntax

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

declaration-list ::=
 declaration
 declaration-list declaration

statement-list ::=
 statement
 statement-list statement
```

Description

A **compound statement** allows you to group statements together in a block of code and use them as if they were a single statement.

Variables and constants declared in the block are local to the block and to any inner blocks unless declared `extern`. If the variables are initialized, the initialization is performed each time the compound statement is entered from the top through the left brace (`{`) character. If the statement is entered via a `goto` statement or in a `switch` statement, the initialization is not performed.

Any variable declared with static storage scope is created and initialized when the program is loaded for execution. This is true even if the variable is declared in an inner block.

Example

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

In this example, variable `temp` is local to the compound statement. It can only be accessed within the compound statement.

Expression and Null Statements

Syntax

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

Description

C **expressions** can be statements. A **null statement** is simply a semicolon by itself.

You can use any valid expression as an expression statement by terminating it with a semicolon. Expression statements are evaluated for their side effects; such as assignment or function calls. If the expression is not specified, but the semicolon is still provided, the statement is treated as a null statement.

Null statements are useful for specifying no-operation statements. No-operation statements are often used in looping constructs where all of the work of the statement is done without an additional statement.

Example

A program fragment that sums up the contents of an array named `x` containing 10 integers might look like this:

```
for(i=0,s=0; i<10; s+=x[i++]);
```

The syntax of the for statement requires a statement following the closing `)` of the `for`. A null statement (;) satisfies this syntax requirement.

Labeled Statements

Syntax

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

Description

Labeled statements are those preceded by a name or tag. You can prefix any statement using a label so at some point you can reference it using `goto` statements. Any statement can have one or more labels.

The `case` and `default` labels can only be used inside a `switch` statement.

Example

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

The `return` statement is labeled `get_out`.

Looping Statements

Syntax

```
while (expression) statement
do statement while (expression);
for ([expression1] ; [expression2]; [expression3])
      statement
```

Description

Use looping statements to force a program to repeatedly execute a statement. The executed statement is called the *loop body*. Loops execute until the control is satisfied. The controlling expression may be any scalar data type.

C has several looping statements: `while`, `do…while`, and `for`. The main difference between these statements is the point at which each loop tests for the exit condition. Refer to the `goto`, `continue`, and `break` statements for ways to exit a loop without reaching its end or meeting loop exit tests.

Examples

The following loops all accomplish the same thing (they assign `i` to `a[i]` for i from 0 to 4):

```
i = 0;
while (i < 5)
{
   a[i] = i;
   i++;
}
```

```
i = 0;
do
{
   a[i] = i;
   i++;
} while (i < 5);
```

```
for (i = 0; i < 5; i++)
{
   a[i] = i;
}
```

Selection Statements

Syntax

```
if (expression) statement [else statement]
switch (expression) statement
```

Description

A selection statement alters a program's execution flow by selecting one path from a collection based on a specified controlling expression. The `if` statement and the `switch` statement are selection statements.

Examples

```
if (expression) statement:
    if (x<y) x=y;

if (expression) statement else statement:
    if (x<y) x=y; else y=x;

switch (expression) statement:
    switch (x)
    { case 1: x=y;
              break;
      default: y=x;
              break;
    }
```

break

Syntax

```
break;
```

Description

A `break` statement terminates the execution of the most tightly enclosing `switch` statement or `for`, `while`, `do`…`while` loop.

Control passes to the statement following the `switch` or iteration statement. You cannot use a `break` statement unless it is enclosed in a `switch` or loop statement. Further, a `break` only exits out of one level of `switch` or loop statement. To exit from more than one level, you must use a `goto` statement.

When used in the `switch` statement, `break` normally terminates each `case` statement. If you do not use `break` (or other unconditional transfer of control), each statement labeled with `case` flows into the next. Although not required, a `break` is usually placed at the end of the last `case` statement. This reduces the possibility of errors when inserting additional cases at a later time.

Example

The following example uses `break` to exit from the `for` loop after executing the loop three times:

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

This example prints:

```
     0
     1
     2
```

continue

Syntax

```
continue;
```

Description

The `continue` statement halts execution of its enclosing `for`, `while`, or `do/while` loop and skips to the next iteration of the loop. In the `while` and `do/while`, this means the expression is tested immediately, and in the `for` loop, the third expression (if present) is evaluated.

Example

```
/*  Program name is "continue_example".  This program
 *  reads a file of student names and test scores.  It
 *  averages each student's grade.  The for loop uses
 *  a continue statement so that the third test score
 *  is not included.
 */
#include <stdio.h>

int main(void)
{
    int test_score, tot_score, i;
    float average;
    FILE *fp;
    char fname[10], lname[15];

    fp = fopen("grades_data", "r");
    while (!feof(fp)) /* while not end of file */
    {
        tot_score = 0;
        fscanf(fp, "%s %s", fname, lname);
        printf("\nStudent's name: %s %s\nGrades: ", fname, lname);
        for (i = 0; i < 5; i++)
        {
            fscanf(fp, "%d", &test_score);
            printf("%d ", test_score);
            if (i == 2)   /*  leave out this test score  */
                continue;
            tot_score += test_score;
        } /*  end for i  */
        fscanf(fp, "\n");  /*  read end-of-line at end of  */
                           /*  each student's data         */
        average = tot_score/4.0;
        printf("\nAverage test score: %4.1f\n", average);
    } /*  end while  */
    fclose(fp);
}
```

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

```
Student's name: Barry Quigley
Grades: 85 91 88 100 75
Average test score: 87.8

Student's name: Pepper Rosenberg
Grades: 91 76 88 92 88
Average test score: 86.8

Student's name: Sue Connell
Grades: 95 93 91 92 89
Average test score: 92.2
```

do...while

Syntax

```
do
      statement;
while (expression);
```

Arguments

statement	A null statement, simple statement, or compound statement.
expression	Any expression.

Description

The `do` statement executes statements within a loop until a specified condition is satisfied. This is one of the three looping constructions in C. Unlike the `for` and `while` loops, `do...while` performs *statement* first and then tests *expression*. If *expression* evaluates to nonzero (true), *statement* executes again, but when *expression* evaluates to zero (false), execution of the loop stops. This type of loop is always executed at least once.

You can jump out of a `do...while` loop prematurely (that is, before *expression* becomes false) by doing the following:

- Use `break` to transfer control to the first statement following the `do...while` loop.
- Use `goto` to transfer control to some labeled statement outside of the loop.
- Use a `return` statement.

Example

```
/*  Program name is "do.while_example". This program finds the
 *  summation (that is, n*(n+1)/2) of an integer that a user
 *  supplies and the summation of the squares of that integer.
 *  The use of the do/while means that the code inside the loop
 *  is always executed at least once.
 */
#include <stdio.h>
int main(void)
```

```
{
    int num, sum, square_sum;
    char answer;

    printf("\n");
    do
    {
        printf("Enter an integer: ");
        scanf("%d", &num);
        sum = (num*(num+1))/2;
        square_sum = (num*(num+1)*(2*num+1))/6;
        printf("The summation of %d is: %d\n", num, sum);
        printf("The summation of its squares is: %d\n",
        square_sum);
        printf("\nAgain? ");
        fflush(stdin);
        scanf("%c", &answer);
    } while ((answer != 'n') && (answer != 'N'));
}
```

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

```
Enter an integer: 10
The summation of 10 is: 55
The summation of its squares is: 385

Again? y
Enter an integer: 25
The summation of 25 is: 325
The summation of its squares is: 5525

Again? n
```

for

Syntax

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

Arguments

expression1	This is the *initialization expression* that typically specifies the initial values of variables. It is evaluated only once before the first iteration of the loop.
expression2	This is the *controlling expression* that determines whether or not to terminate the loop. It is evaluated before each iteration of the loop. If *expression2* evaluates to a nonzero value, the loop body is executed. If it evaluates to 0, execution of the loop body is terminated and control passes to the first statement after the loop body. This means that if the initial value of *expression2* evaluates to zero, the loop body is never executed.
expression3	This is the *increment expression* that typically increments the variables initialized in *expression1*. It is evaluated after each iteration of the loop body and before the next evaluation of the controlling expression.

Description

The `for` statement executes the statements within a loop as long as *expression2* is true. The `for` statement is a general-purpose looping construct that allows you to specify the initialization, termination, and increment of the loop. The `for` uses three expressions. Semicolons separate the expressions. Each expression is optional, but you must include the semicolons.

How the for Loop is Executed

The `for` statement works as follows:

1. First, *expression1* is evaluated. This is usually an assignment expression that initializes one or more variables.
2. Then *expression2* is evaluated. This is the conditional part of the statement.
3. If *expression2* is false, program control exits the `for` statement and flows to the next statement in the program. If *expression2* is true, *statement* is executed.

4. After *statement* is executed, *expression3* is evaluated. Then the statement loops back to test *expression2* again.

for Loop Processing

The `for` loop continues to execute until *expression2* evaluates to 0 (false), or until a branch statement, such as a `break` or `goto`, interrupts loop execution.

If the loop body executes a `continue` statement, control passes to *expression3*. Except for the special processing of the `continue` statement, the `for` statement is equivalent to the following:

```
expression1;
while (expression2) {
   statement
   expression3;
}
```

You may omit any of the three expressions. If *expression2* (the controlling expression) is omitted, it is taken to be a nonzero constant.

for versus while Loops

Note that for loops can be written as while loops, and vice versa. For example, the for loop

```
for (j = 0; j < 10; j++)
{
  do_something();
}
```

is the same as the following while loop:

```
j = 0;
while (j<10)
{
 do_something();
 j++;
}
```

Example

```
/*  Program name is "for_example". The following computes a
 *  permutation  that is, P(n,m) = n!/(n-m)!  using for
 *  loops to compute n! and (n-m)!
 */
#include <stdio.h>
```

```
#define SIZE 10

int main(void)
{
    int n, m, n_total, m_total, perm, i, j, mid, count;

    printf("Enter the numbers for the permutation (n things ");
    printf("taken m at a time)\nseparated by a space: ");
    scanf("%d %d", &n, &m);
    n_total = m_total = 1;
    for (i = n; i > 0; i--)        /*  compute n!  */
        n_total *= i;
    for (i = n - m; i > 0; i--)    /*  compute (n-m)!  */
        m_total *= i;
    perm = n_total/m_total;
    printf("P(%d,%d) = %d\n\n", n, m, perm);

/*  This series of for loops prints a pattern of "Z's" and shows
 *  how loops can be nested and how you can either increment or
 *  decrement your loop variable. The loops also show the proper
 *  placement of curly braces to indicate that the outer loops
 *  have multiple statements.
 */
    printf("Now, print the pattern three times:\n\n");
    mid = SIZE/2;

/*  controls how many times pattern is printed */
    for (count = 0; count < 3; count++)
    {
        for (j = 0; j < mid; j++)
        {
        /*  loop for printing an individual line  */
            for (i = 0; i < SIZE; i++)
                if (i < mid - j || i > mid + j)
                    printf(" ");
                else
                    printf("Z");
            printf("\n");
        }
    for (j = mid; j >= 0; j--)
    {
            for (i = 0; i <= SIZE; i++)
```

```
                if (i < mid - j || i > mid + j)
                    printf(" ");
                else
                    printf("Z");
              printf("\n");
     }
   }
}
```

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

```
Enter the numbers for the permutation (n things taken m at a
time) separated by a space: 4 3
P(4,3) = 24

Now, print the pattern three times:

     Z
    ZZZ
   ZZZZZ
  ZZZZZZZ
 ZZZZZZZZZ
ZZZZZZZZZZZ
 ZZZZZZZZZ
  ZZZZZZZ
   ZZZZZ
    ZZZ
     Z
     Z
    ZZZ
   ZZZZZ
  ZZZZZZZ
 ZZZZZZZZZ
ZZZZZZZZZZZ
 ZZZZZZZZZ
  ZZZZZZZ
   ZZZZZ
    ZZZ
     Z
     Z
    ZZZ
   ZZZZZ
  ZZZZZZZ
 ZZZZZZZZZ
ZZZZZZZZZZZ
 ZZZZZZZZZ
  ZZZZZZZ
   ZZZZZ
    ZZZ
     Z
```

goto

Syntax

`goto` *`label`*;

Arguments

`label`	This is a label or tag associated with an executable statement.

Description

The purpose of the `goto` statement is to enable program control to jump to some other statement. The destination statement is identified by a statement label, which is just a name followed by a colon. The label must be in the same function as the `goto` statement that references it.

Few programming statements have produced as much debate as the `goto` statement. The `goto` statement is necessary in more rudimentary programming languages, but its use in high-level languages is generally frowned upon. Nevertheless, most high-level languages, including C, contain a `goto` statement for those rare situations where it can't be avoided.

With deeply nested logic there are times when it is cleaner and simpler to bail out with one `goto` rather than backing out of the nested statements. The most common and accepted use for a `goto` is to handle an extraordinary error condition.

Example

The following example shows a `goto` that can easily be avoided by using the `while` loop, and also shows an illegal `goto`:

```
/*  Program name is "goto_example". This program finds the
 *  circumference and area of a circle when the user gives
 *  the circle's radius.
 */
#include <stdio.h>
#define PI 3.14159

int main(void)
{
    float cir, radius, area;
    char answer;
    extern void something_different(void);

circles:
    printf("Enter the circle's radius: ");
    scanf("%f", &radius);
    cir = 2 * PI * radius;
    area = PI * (radius * radius);
    printf("The circle's circumference is: %6.3f\n", cir);
    printf("Its area is: %6.3f\n", area);
    printf("\nAgain? y or n: ");
    fflush(stdin);
    scanf("%c", &answer);

    if (answer == 'y' || answer == 'Y')
        goto circles;
    else {
        printf("Do you want to try something different? ");
        fflush(stdin);
        scanf("%c", &answer);
        if (answer == 'y' || answer == 'Y')
    /*      goto different;       WRONG! This label is in  */
    /*                            another block.           */
        something_different();
    } /*  end else  */
}

void something_different(void)
{
different:
    printf("Hello. This is something different.\n");
}
```

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

```
Enter the circle's radius: 3.5
The circle's circumference is: 21.991
Its area is: 38.484

Again? y or n: y
Enter the circle's radius: 6.1
The circle's circumference is: 38.327
Its area is: 116.899
```

```
Again? y or n: n
Do you want to try something different? y
Hello. This is something different.
```

if

Syntax

```
if (exp)      /* format 1 */
    statement

if (exp)      /* format 2 */
    statement1
else
    statement2
```

Arguments

exp	Any expression.
statement	Any null statement, simple statement, or compound statement. A statement can itself be another `if` statement. Remember, a statement ends with a semicolon.

Description

The `if` statement tests one or more conditions and executes one or more statements according to the outcome of the tests. The `if` and `switch` statements are the two conditional branching statements in C.

In the first form, if *exp* evaluates to true (any nonzero value), C executes *statement*. If *exp* is false (evaluates to 0), C falls through to the next line in the program.

In the second form, if *exp* evaluates to true, C executes *statement1*, but if *exp* is false, *statement2* is performed.

A statement can be an `if` or `if…else` statement.

Example 1

You can test multiple conditions with a command that looks like this:

```
if (exp1)      /*  multiple conditions */
    statement1
else if (exp2)
    statement2
```

```
else if (exp3)
    statement3
.. .
else
   statement N
```

The important thing to remember is that C executes at most only one statement in the `if…else` and `if…else/if…else` constructions. Several expressions may indeed be true, but only the statement associated with the first true expression is executed.

Example 2

Expressions subsequent to the first true expression are not evaluated. For example:

```
/* determine reason the South lost the American Civil War */
if (less_money)
    printf("It had less money than the North.\n");
else if (fewer_supplies)
    printf("It had fewer supplies than the North.\n");
else if (fewer_soldiers)
    printf("It had fewer soldiers.\n");
else
{
    printf("Its agrarian society couldn't compete with the ");
    printf("North's industrial one.\n");
}
```

All the expressions in the above code fragment could be evaluated to true, but the run-time system would only get as far as the first line and never even test the remaining expressions.

Using Braces in Compound if Statements

Use curly braces (`{ }`) in a compound statement to indicate where the statement begins and ends. For example:

```
if (x > y) {
    temp = x;
    x = y;
    y = temp;
}
else
    /* make next comparison */
```

Braces also are important when you nest if statements. Since the else portion of the statement is optional, you may not have one for an inner if. However, C associates an else with the closest previous if statement unless you use braces to show that isn't what you want. For example:

```
if (month == 12) {     /* month = December */
    if (day == 25)
        printf("Today is Christmas.\n");
}
else
    printf("It's not even December.\n");
```

Without the braces, the else would be associated with the inner if statement, and so the no-December message would be printed for any day in December except December 24. Nothing would be printed if month did not equal 12.

The Dangling else

Nested if statements create the problem of matching each else phrase to the right if statement. This is often called the dangling else problem; the general rule is:

- An else is always associated with the nearest previous if.

Each if statement, however, can have only one else clause. It is important to format nested if statements correctly to avoid confusion. An else clause should always be at the same indentation level as its associated if. However, do not be misled by indentations that look right even though the syntax is incorrect.

Example

```
/*  Program name is "if.else_example". */
#include <stdio.h>
int main(void)
{
    int age, of_age;
    char answer;
/*  This if statement is an example of the second form (see
 *  "Description" section).  */
    printf("\nEnter an age: ");
    scanf("%d", &age);
    if (age > 17)
        printf("You're an adult.\n");
    else {
        of_age = 18 - age;
        printf("You have %d years before you're an adult.\n",
                  of_age);
    } /*  end else  */
    printf("\n");
    printf("This part will help you decide whether to jog \
 today.\n");
    printf("What is the weather like?\n");
    printf("      raining = r\n");
    printf("      cold = c\n");
    printf("      muggy = m\n");
    printf("      hot = h\n");
    printf("      nice = n\n");
    printf("Enter one of the choices: ");
    fflush(stdin);
    scanf("\n%c", &answer);
/*  This if statement is an example of the third form (see
 *  "Description" section.  */
    if (answer == 'r')
        printf("It's too wet to jog today. Don't bother.\n");
    else if (answer == 'c')
        printf("You'll freeze if you jog today. Stay indoors.\n");
    else if (answer == 'm')
        printf("It's no fun to run in high humidity. Skip it.\n");
    else if (answer == 'h')
        printf("You'll die of the heat if you try to jog today. \
 So don't.\n");
    else if (answer == 'n')
        printf("You don't have any excuses. You'd better go \
 run.\n");
    else
        printf("You didn't give a valid answer.\n");
}
```

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

```
Enter an age: 15
You have 3 years before you're an adult.

This part will help you decide whether to jog today.
What is the weather like?
      raining = r
      cold = c
      muggy = m
      hot = h
      nice = n
Enter one of the choices: r
It's too wet to jog today. Don't bother.
```

return

Syntax

```
return;             /* first form */
return exp;         /* second form */
```

Arguments

exp	Any valid C expression.

Description

The `return` statement causes a C program to exit from the function containing the `return` and go back to the calling block. It may or may not have an accompanying *exp* to evaluate. If there is no *exp*, the function returns an unpredictable value.

A function may contain any number of return statements. The first one encountered in the normal flow of control is executed, and causes program control to be returned to the calling routine. If there is no `return` statement, program control returns to the calling routine when the right brace of the function is reached. In this case, the value returned is undefined.

Return Types

The return value must be assignment-compatible with the type of the function. This means that the compiler uses the same rules for allowable types on either side of an assignment operator to determine allowable return types. For example, if `f()` is declared as a function returning an `int`, it is legal to return any arithmetic type, since they can all be converted to an `int`. It would be illegal, however, to return an aggregate type or a pointer, since these are incompatible types.

The following example shows a function that returns a `float`, and some legal return values.

```
float f(void)
{
    float f2;
    int a;
    char c;
    f2 = a;      /* OK, quietly converts a to float */
    return a;    /* OK, quietly converts a to float */
    f2 = c;      /* OK, quietly converts c to float */
    return c;    /* OK, quietly converts c to float */
}
```

Pointer Return Types

The C language is stricter about matching pointers. In the following example, `f()` is declared as a function returning a pointer to a `char`. Some legal and illegal return statements are shown.

```
char *f(void)
{
  char **cpp, *cp1, *cp2, ca[10];
  int *ip1, *ip2;

  cp1 = cp2;        /* OK, types match */
  return cp2;       /* OK, types match */
  cp1 = *cpp;       /* OK, types match */
  return *cpp;      /* OK, types match */

/* An array name without a subscript is converted
 * to a pointer to the first element.
 */
  cp1 = ca;      /* OK, types match */
  return ca;     /* OK, types match */

  cp1 = *cp2;    /* Error, mismatched types         */
                 /* (pointer to char vs. char)      */
  return *cp2;   /* Error, mismatched types         */
                 /* (pointer to char vs. char)      */
  cp1 = ip1;     /* Error, mismatched pointer types */
  return ip1;    /* Error, mismatched pointer types */
  return;        /* Produces undefined behavior     */
                 /* should return (char *)          */
}
```

Note in the last statement that the behavior is undefined if you return nothing. The only time you can safely use `return` without an expression is when the function type is `void`. Conversely, if you return an expression for a function that is declared as returning `void`, you will receive a compile-time error.

Functions can return only a single value directly via the `return` statement. The `return` value can be any type except an array or function. This means that it is possible to return more than a single value indirectly by passing a pointer to an aggregate type. It is also possible to return a structure or union directly. HP C implements this by passing the structure or union by reference if the structure or union is greater than eight bytes.

Example

```
/*  Program name is "return_example".
 *  This program finds the length of a word that is entered.
 */
#include <stdio.h>

int find_length( char *string )
{
    int i;
    for (i =0; string[i] != '\0'; i++);
    return i;
}

int main( void )
{
    char string[132];
    int result;
    int again = 1;
    char answer;

    printf( "This program finds the length of any word you ");
    printf( "enter.\n" );
    do
    {
       printf( "Enter the word: ");
       fflush(stdin);
       gets( string );
       result = find_length( string );
       printf( "This word contains %d characters. \n", result);
       printf("Again? ");
       scanf("%c", &answer);
     } while (answer == 'Y' || answer == 'y');
}
```

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

```
This program finds the length of any string you enter.

Enter the string: **Copenhagen**
The string is 10 characters.
Again? **y**

Enter the string: **galaxy**
The string is 6 characters.
Again? **n**
```

switch

Syntax

```
switch ( exp )
{
  case const_exp : [statement]...
[case const_exp : [statement]...]
[default : [statement]...]

}
```

Arguments

exp	The integer expression that the `switch` statement evaluates and then compares to the values in all the cases.
const_exp	An integer expression to which *exp* is compared. If *const_exp* matches *exp*, the accompanying statement is executed.
statement	Zero or more simple statements. (If there is more than one simple statement, you do not need to enclose the statements in braces.)

Description

The `switch` statement is a conditional branching statement that selects among several statements based on constant values.

The expression immediately after the `switch` keyword must be enclosed in parentheses and must be an integral expression.

The expressions following the `case` keywords must be integral constant expressions; that is, they may not contain variables.

An important feature of the `switch` statement is that program flow continues from the selected `case` label until another control-flow statement is encountered or the end of the `switch` statement is reached. That is, the compiler executes any statements following the selected `case` label until a `break`, `goto`, or `return` statement appears. The `break` statement explicitly exits the `switch` construct, passing control to the statement following the `switch` statement. Since this is usually what you want, you should almost always include a `break` statement at the end of the statement list following each case label.

The following print_error() function, for example, prints an error message based on an error code passed to it.

```
/*  Prints error message based on error_code.
 *  Function is declared with void because it doesn't
 *  return anything.
 */
#include <stdio.h>
#define ERR_INPUT_VAL 1
#define ERR_OPERAND 2
#define ERR_OPERATOR 3
#define ERR_TYPE 4
void print_error(int error_code)
{
    switch (error_code) {
        case  ERR_INPUT_VAL:
             printf("Error: Illegal input value.\n");
             break;
        case  ERR_OPERAND:
             printf("Error: Illegal operand.\n");
             break;
        case  ERR_OPERATOR:
             printf("Error: Unknown operator.\n");
             break;
        case  ERR_TYPE:
             printf("Error: Incompatible data.\n");
             break;
        default: printf("Error: Unknown error code %d\n",
                           error_code);
             break;
    }
}
```

The break statements are necessary to prevent the function from printing more than one error message. The last break after the default case is not really necessary, but it is a good idea to include it anyway for the sake of consistency.

Evaluation of switch Statement

The switch expression is evaluated; if it matches one of the case labels, program flow continues with the statement that follows the matching case label. If none of the case labels match the switch expression, program flow continues at the default label, if it exists. (The default label need not be the last label, though it is good style to put it last.) No two case labels may have the same value.

Associating Statements with Multiple case Values

Sometimes you want to associate a group of statements with more than one case value. To obtain this behavior, you can enter consecutive case labels. The following function, for instance, returns 1 if the argument is a punctuation character, or 0 if it is anything else.

```
/* This function returns 1 if the argument is a
 * punctuation character. Otherwise, it returns 0.
 */
is_punc(char arg)
{
    switch (arg) {
        case '.':
        case ',':
        case ':':
        case ';':
        case '?':
        case '-':
        case '(':
        case ')':
        case '!':  return 1;
        default :  return 0;
    }
}
```

Example

```
/* Use the switch statement to decide which comment should be printed */
#include <stdio.h>
int main(void)
{
    char answer, grade;
    answer = 'y';
    printf("\n\n");
    while (answer == 'y' || answer == 'Y') {
         printf("Enter student's grade: ");
         fflush(stdin);
         scanf("%c", &grade);
         printf("\nComments: ");
         switch (grade) {
              case 'A':
              case 'a':
                   printf("Excellent\n");
```

```
                    break;
               case 'B':
               case 'b':
                    printf("Good\n");
                    break;
               case 'C':
               case 'c':
                    printf("Average\n");
                    break;
               case 'D':
               case 'd':
                    printf("Poor\n");
                    break;
               case 'E':
               case 'e':
               case 'F':
               case 'f':
                    printf("Failure\n");
                    break;
               default:
                    printf("Invalid grade\n");
                    break;
          }  /* end switch */
          printf("\nAgain? ");
          fflush(stdin);
          scanf("%s", &answer);
     }
}
```

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

```
Enter student's grade: B

Comments: Good

Again? y
Enter student's grade: C

Comments: Average

Again? n
```

while

Syntax

```
while ( exp )
        statement
```

Arguments

exp	Any expression.
statement	This statement is executed when the `while` (*exp*) is true.

Description

The `while` statement executes the statements within a loop as long as the specified condition, *exp*, is true. This is one of the three looping constructions available in C. Like the `for` loop, the `while` statement tests *exp* and if it is true (nonzero), *statement* is executed. Once *exp* becomes false (0), execution of the loop stops. Since *exp* could be false the first time it is tested, *statement* may not be performed even once.

The following describes two ways to jump out of a `while` loop prematurely (that is, before *exp* becomes false):

- Use `break` to transfer control to the first statement following the while loop.
- Use `goto` to transfer control to some labeled statement outside the loop.

Example

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

int main(void)
{
    int count = 0, count2 = 0;
    char a_string[80], *ptr_to_a_string = a_string;

    printf("Enter a string -- ");
    gets(a_string);
    while (*ptr_to_a_string++)
        count++;   /* A simple statement loop */
    printf("The string contains %d characters.\n", count);
    printf("The first word of the string is ");

    while (a_string[count2] != ' ' && a_string[count2] != '\0')
    {
    /* A compound statement loop */
        printf ("%c", a_string[count2]);
        count2++;
    }
    printf("\n");
}
```

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

```
Enter a string Four score and seven years ago
The string contains 30 characters.
The first word of the string is Four
```

7 Preprocessing Directives

Preprocessing directives work as compiler control lines. They allow you to direct the compiler to perform certain actions on the source file.

You can select from any of the following topics:

- Source File Inclusion (#include)
- Macro Replacement (#define, #undef)
- Predefined Macros
- Conditional Compilation (#if, #ifdef, ..#endif)
- Line Control (#line)
- Pragma Directive (#pragma)
- _Pragma
- Error Directive (#error)
- Trigraph Sequences

Overview of the Preprocessor

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

Syntax

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

Description

The preprocessing directives control the following general functions:

1. **Source File Inclusion**

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

2. **Macro Replacement**

   You can direct the compiler to replace token sequences with other token sequences. This is frequently used to define names for constants rather than hard coding them into the source files.

3. **Conditional Inclusion**

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

4. **Line Control**

   You can direct the compiler to increment subsequent lines from a number specified in a control line.

5. **Pragma Directive**

Pragmas are implementation-dependent instructions that are directed to the compiler. Because they are very system dependent, they are not portable.

All preprocessing directives begin with a pound sign (#) as the first character in a line of a source file. White space may precede the # character in preprocessing directives. The # character is followed by any number of spaces and horizontal tab characters and the preprocessing directive. The directive is terminated by a new-line character. You can continue directives, as well as normal source lines, over several lines by ending lines that are to be continued with a backslash (\).

Comments in the source file that are not passed through the preprocessor are replaced with a single white-space character.

Examples

```
include-directive:     #include <stdio.h>

macro-directive:      #define MAC x+y

conditional-directive: #ifdef MAC

line-directive:       #line 5 "myfile"

pragma-directive:    #pragma INTRINSIC func
```

Source File Inclusion (#include)

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

Syntax

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

Description

In the third form above, *`identifier`* must be in the form of one of the first two choices after macro replacement.

The `#include` preprocessing directive causes the compiler to switch its input file so that source is taken from the file named in the include directive. Historically, include files are named:

```
  filename.h
```

If the file name is enclosed in double quotation marks, the compiler searches your current directory for the specified file. If the file name is enclosed in angle brackets, the “system” directory is searched to find the named file. Refer to Chapter 10 for a detailed description of how the directory is searched.

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

The arguments to the `#include` directive are subject to macro replacement before the directive processes them. Error messages produced by the HP C compiler usually supply the file name the error occurred in as well as the file relative line number of the error.

Examples

```
#include <stdio.h>

#include "myheader"

#ifdef   MINE
#   define   filename   "file1"
```

```
#else
#   define  filename  "file2"
#endif

#include filename
```

Macro Replacement (#define, #undef)

You can define text substitutions in your source file with C macro definitions.

Syntax

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

replacement-list ::=
    token
    replacement-list token
```

Description

A `#define` preprocessing directive of the form:

`#define identifier [replacement-list]`

defines the *identifier* as a macro name that represents the replacement list. The macro name is then replaced by the list of tokens wherever it appears in the source file (except inside of a string or character constant, or comment). A macro definition remains in force until it is undefined through the use of the `#undef` directive or until the end of the translation unit.

Macros can be redefined without an intervening `#undef` directive. Any parameters used must agree in number and spelling, and the replacement lists must be identical. All white space is treated equally.

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

If the define takes the form

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

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

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

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

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

After all replacements have taken place during macro invocation, each instance of the special ## token is deleted and the tokens preceding and following the ## are concatenated into a single token. This is useful in forming unique variable names within macros.

The following example illustrates the use of the # operator for creating string literals out of arguments and concatenating tokens:

```
#define debug(s, t) printf("x" # s "= %d, x" # t " %s", x ## s, x ## t)
```

Invoked as: `debug(1, 2);`

Results in:

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
```

which, after concatenation, results in:

```
printf("x1= %d, x2= %s", x1, x2);
```

Spaces around the # and ## are optional.

NOTE The # and ## operators are only supported in ANSI mode.

The most common use of the macro replacement is in defining a constant. Rather than hard coding constants in a program, you can name the constants using macros then use the names in place of actual constants. By changing the definition of the macro, you can more easily change the program:

```
#define ARRAY_SIZE 1000

float x[ARRAY_SIZE];
```

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

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

Changing the dimension of `x` means only changing the macro for `ARRAY_SIZE`; the dimension will change and so will all the expressions that make use of the dimension.

Some other common macros used by C programmers include:

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

The following macro is more complex. It has two parameters and will produce an in-line expression which is equal to the maximum of its two parameters:

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

Parentheses surrounding each argument and the resulting expression insure that the precedences of the arguments and the result will not improperly interact with any other operators that might be used with the `MAX` macro.

Using a macro definition for `MAX` has some advantages over a function definition. First, it executes faster because the macro generates in-line code, avoiding the overhead of a function call. Second, the `MAX` macro accepts any argument types. A functional implementation of `MAX` would be restricted to the types defined for the function. Note further that because each argument to the `MAX` macro appears in the token string more than once, check to be sure that the actual arguments to the `MAX` macro do not have any "side effects." The following example

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

might not work as expected because the argument `a` is incremented two times when `a` is the maximum.

The following statement

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

is expanded to:

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

Examples

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

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

Predefined Macros

In addition to `__LINE__` and `__FILE__` (see “Line Control (#line)” on page 198), ANSI C provides the `__DATE__`, `__TIME__` and `__STDC__` predefined macros. Table 7-1 describes the complete set of macros that are predefined to produce special information. They may not be undefined.

Table 7-1 Predefined Macros

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

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

Conditional Compilation (#if, #ifdef, ..#endif)

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

Syntax

```
conditional-directive ::=
    #if     constant-expression newline [group]
    #ifdef  identifier newline [group]
    #ifndef identifier newline [group]
    #else   newline [group]
    #elif   constant-expression newline [group]
    #endif
```

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

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

Description

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

```
#if constant-expression
 .
 .
 .
/* (Code that compiles if the expression evaluates
    to a nonzero value.) */
#else
 .
 .
 .

 .
/* (Code that compiles if the expression evaluates
    to a zero value.) */
#endif
```

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

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

The parentheses around the identifier are optional. For example:

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

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

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

The `#if` preprocessing directive has the form:

```
  #if constant-expression
```

Use `#if` to test an expression. The compiler evaluates the expression in the directive. If it is true (a nonzero value), the code following the directive is included. If the expression evaluates to false (a zero value), the compiler ignores the code up to the next `#else`, `#endif`, or `#elif` directive.

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

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

The following preprocessing directives are used to test for a definition:

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

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

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

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

The #elif *constant-expression* directive tests whether a condition of the previous #if, #ifdef, or #ifndef was false. #elif is syntactically the same as the #if directive and can be used in place of an #else directive.

Examples

Valid combinations of these conditional compilation directives follow:

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

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

#if A>47
                  /* compiled if A evaluates > 47 */
#else
#if A < 20
                  /* compiled if A evaluates < 20 */
#else
                  /* compiled if A >= 20 and <= 47 */
#endif            /* end of if, A < 20 */
#endif             /* end of if, A > 47 */
```

Examples

```
#ifdef (HP9000_S800)    /* If HP9000_S800 is defined, INT_SIZE */
#define INT_SIZE 32     /* is defined to be 32 (bits).  */
#elif defined (HPVECTRA) && defined (SMALL_MODEL)
#define INT_SIZE 16     /* Otherwise, if HPVECTRA and */
#endif                  /* SMALL_MODEL are defined,INT_SIZE is */

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

Line Control (#line)

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

Syntax

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

Description

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

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

Example

```
#line digit-sequence [filename]:  #line 5 "myfile"
```

Pragma Directive (#pragma)

You can provide instructions to the compiler through inclusion of pragmas.

Syntax

```
pragma-directive ::=
#pragma replacement-list
```

Description

The `#pragma` preprocessing directive provides implementation-dependent information to the compiler. See Chapter 9, “Compiling and Running HP C Programs,” on page 207 for descriptions of pragmas recognized by HP C/HP-UX. Any pragma that is not recognized by the compiler is ignored.

Example

```
#pragma OPTIMIZE ON
```

```
#pragma OPTIMIZE OFF
```

This pragma is used to toggle optimization on/off for different sections of source code as these pragmas are encountered in a top to bottom read of a source file.

_Pragma

_Pragma is a preprocessing unary operator.

Syntax

```
_Pragma (string-literal)
```

_Pragma is a new preprocessing operator and is a part of C99 standards. The string literal is destringized by deleting the `L` prefix, if present. _Pragma deletes the leading and trailing double quotes, replacing each escape sequence by a double-quote, and replacing the escape sequence by a single backlash. The resulting sequence of characters is processed to produce preprocessor tokens that are executed as if they were preprocessed tokens in a pragma directive.

The `_Pragma` operator provides portability in the use of an existing `#pragma` C preprocessor construct. These pragmas are processed as defined in its implementation. Most C implementations provide pragmas that are very similar in meaning and functionality. The `_Pragma` operator can be used in the replacement text of a macro, so as to aid in abstracting these specific pragmas a level higher.

Examples

The following examples list the usage of the `_Pragma` operator:

Example 7-1 _Pragma Operator Usage

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 as earlier, if it appears literally as shown or it results from macro replacement, as in:

```
# define LISTING(x) PRAGMA(listing on #x)
# define PRAGMA(x) _Pragma(#x)
LISTING (..\listing.dir)
```

Example 7-2 _Pragma Operator Usage

```
_Pragma("ALIGN 4")
```

expands to

```
#pragma ALIGN 4
```

Example 7-3 _Pragma Operator Usage

```
#define FOO "ALIGN 4"
_Pragma(FOO)
```

expands to

```
#pragma ALIGN 4
```

Error Directive (#error)

Syntax

```
#error [pp-tokens]
```

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

Examples

```
#ifndef (HP_C)
#error "HP_C not defined!"      /* This directive will produce
#endif                             the diagnostic message "HP_C
                                    not defined!"              */

#if TABLE_SIZE % 256 != 0
#error "TABLE_SIZE must be a multiple of 56!"
#endif                          /* This directive will produce
                                   the diagnostic message
                                   "TABLE_SIZE must be a
                                    multiple of 256!          */
```

NOTE The `#error` directive is only supported in ANSI mode.

Trigraph Sequences

The C source code character set is a superset of the ISO 646-1983 Invariant Code Set. To enable programs to be represented in the reduced set, *trigraph sequences* are defined to represent those characters not in the reduced set. A *trigraph* is a three character sequence that is replaced by a corresponding single character. Table 7-2 gives the complete list of trigraph sequences and their replacement characters.

Table 7-2 Trigraph Sequences and Replacement Characters

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

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

-notrigraph Option

In the ANSI extended (`-Ae`) mode, trigraph translation is automatically done. The option, `-notrigraph` will disable automatic translation of trigraph sequences.

8 C Library Functions

The C library (`/usr/lib/hpux32/libc.so` or `/usr/lib/hpux64/libc.so`) is divided into different subsections. Each subsection has a header file that defines the objects found in that section of the library.

The standard headers are:

```
<assert.h>     <locale.h>    <stddef.h>
<ctype.h>      <math.h>      <stdio.h>
<errno.h>      <setjmp.h>    <stdlib.h>
<float.h>      <signal.h>    <string.h>
<limits.h>     <stdarg.h>    <time.h>
```

The order of inclusion of these header files using the `#include` directive makes no difference. Also, if you include the same header file more than once, an error does not occur.

Function names beginning with an underscore `(_)` are reserved for library use; you should not specify identifiers that begin with an underscore.

To use some facilities, the C source code must include the preprocessor directive:

```
#include <libraryname.h>
```

The preprocessor looks for the particular header file defined in *`libraryname`* in a standard location on the system.

The standard location is `/usr/include`.

The *`libraryname`* must be enclosed in angle brackets. For example, if you want to use the `fprintf` function, which is in the standard I/O library, your program must specify

```
#include <stdio.h>
```

because the definition of `fprintf,` as well as various types and variables used by the I/O function, are found in the `stdio.h` header file.

The C library contains both functions and macros. The use of macros improves the execution speed of certain frequently used operations. One drawback to using macros is that they do not have an address. For example, if a function expects the address of (a pointer to) another function as an argument, you cannot use a macro name in that argument. The following example illustrates the drawback:

```
#define add1(x) ((x)+=1)

extern f();

main()
```

```
     {
          .
          .
          .
          f(add1); <This construct is illegal.
          .
          .
          .
       }
```

Using `add1` as an argument causes an error.

The `#undef` directive may be used to reference a true function instead of a macro.

There are three ways in which a function can be used:

- In a header file (which might generate a macro)

```
#include <string.h>
i = strlen(x);
```

- By explicit declaration

```
extern int strlen();
i=strlen(x);
```

- By implicit declaration

```
i = strlen(x);
```

NOTE It is recommended you always include a header to declare C library functions.

For more information on C library functions, see the *HP-UX Reference* and *HP-UX Linker and Libraries User Guide*.

9 Compiling and Running HP C Programs

This chapter describes how to compile and run HP C programs. The compiler command and its options are presented. You can compile HP C programs to produce assembly, object, or executable files. You can also optionally optimize code to improve application run-time speed.

Compiling HP C Programs

When you compile a program, it passes through one or more of the following steps depending upon which command line options you use:

- **Preprocessor:** This phase examines all lines beginning with a `#` and performs the corresponding actions and macro replacements.
- **Compilation Process:** This phase takes the output of the preprocessor and generates object code.
- **Optimization:** This optional phase optimizes the generated object code.
- **Linking:** In this phase, the linker is invoked to produce an executable program. External references in shared and archived libraries are resolved as required. The startup routines are copied in, and the C library in `/usr/lib/hpux##/lib.so` (## is 32 or 64) is referenced. (By default, shared libraries take precedence over archived libraries if both versions are available. However, if you use the `LPATH` environment variable, you should make sure that all shared libraries come before any archive library directories. See the *HP-UX Linker and Libraries Online User Guide* for information on `LPATH` or on creating and linking with shared libraries.) Object modules are combined into an executable program file.

The cc(1) Command

Use the `cc(1)` command to compile HP C programs. It has the following format:

`cc [`*`options`*`]` *`files`*

where:

`options`	is one or more compiler options and their arguments, if any. Options can be grouped together under one minus sign.
`files`	is one or more file names, separated by blanks. Each file is either a source or an object file.

Specifying Files to the cc Command

Files with names ending in `.c` are assumed to be HP C source files. Each HP C source file is compiled, producing an object file with the same name as the source file except that the `.c` extension is changed to a `.o` extension. However, if you compile and link a single source file into an HP C program in one step, the `.o` file is automatically deleted.

Files with names ending in `.i` are assumed to be preprocessor output files. Files ending in `.i` are processed the same as `.c` files, except that the preprocessor is not run on the `.i` file before the file is compiled.

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

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

Specifying Options to the cc Command

Each compiler option has the following format:

```
    -optionname [optionarg]
```

where:

optionname	is the name of a standard compiler option.
optionarg	is the argument to *optionname*.

The optional argument -- delimits the end of options. Any following arguments are treated as operands (typically input filenames) even if they begin with the minus (-) character.

An Example of Using a Compiler Option

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

```
cc demo.c
```

the executable file is named `a.out`.

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

cc -o my_executable my_source.c

Concatenating Options

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

For example, suppose you want to compile `my_file.c` using the `-v`, `-g`, and `-DPROG=sub` compiler options. There are several ways to do this:

```
cc my_file.c -v -g -DPROG=sub
cc my_file.c -vg -D PROG=sub

cc my_file.c -vgDPROG=sub
cc -vgDPROG=sub my_file.c
```

HP C Compiler Options

Table 9-1 summarizes the command line options supported by HP Integrity servers. Some of these options are HP aC++ options. Refer to *HP aC++/HP C Programmer's Guide* for detailed information on compiler options.

Table 9-1 HP C Compiler Options at a Glance

Option	Description
`-.suffix`	Directs output from the `-E` option into a file with a corresponding .suffix instead of a `.c` file.
`-Aa`	Enables strict ANSI C compliance.
`-AA`	Enables use of 2.0 Standard C++ library.
`-Aarm`	Enables Tru64 UNIX C++ ARM dialect.
`-AC89`	Enables ANSI C89 compliance.
`-AC99`	Enables ANSI C99 compliance.
`-Ae`	Enables ANSI C89 compliance, HP value-added features (as described for `+e` option), and `_HPUX_SOURCE` name space macro. It is equivalent to `-AC89 +e`.
`-Ag++`	Enables GNU C++ dialiect compatibility.
`-Agcc`	Enables GNU C dialect compatibility.
`-AP`	Turns off the `-AA` mode and uses the older C++ runtime libraries.
`-b`	Creates a shared library rather than an executable file.
`-Bdefault`	Assigns default export class to global symbols.
`-Bhidden`	Assigns hidden export class to symbols.

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

Option	Description
`-Bextern`	Performs the same operation as `+Oextern=sym1,sym2,sym3...` except that symbols are loaded from an existing file, instead of specified on the command line.
`-Bprotected`	Assigns protected export class to symbols.
`-Bprotected_data`	Assigns protected export class to data symbols.
`-Bprotected_def`	Assigns protected export class to locally defined symbols.
`-Bsymbolic`	Assigns protected export class to all symbols.
`-c`	Compiles only, does not link.
`-C`	Prevents the preprocessor from stripping comments.
`-D`*`name`*	Defines the preprocessor variable *`name`* with a value of "1".
`-D`*`name=def`*	Defines the preprocessor variable *`name`* with a value of *`def`*.
`-dynamic`	Enables linking of PIC objects.
`-E`	Performs preprocessing only with output to `stdout`.
`-e epsym`	Sets the default entry point address for output file to the same as the symbol *epsym*.
`-exec`	Indicates that object files created will be used to create an executable file.
`-ext`	Specifying `-ext` enables HP aC++ extension to the C++ standard.
`-fast`	Selects a combination of optimization options for optimum execution speed and reasonable build times.

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

Option	Description
`-[no]fpwidetypes`	Enables [disables] extended and quad floating point data types.
`-g`	Inserts information for the symbolic debugger in the object file.
`-g0`	Causes the compiler to generate complete debug information for the debugger.
`-g1`	Causes the debugger to generate minimal information for the debugger.
`-G`	Inserts information required by the `gprof` profiler in the object file.
`-ipo`	Enables interprocedural optimizations across files.
`-I`*`dir`*	Inserts *`dir`* in the include file search path.
`-lname`	Causes the linker to search one of the default libraries to resolve unresolved external references.
`-l`*`x`*	Links with the `/lib/libx.a` and `/usr/lib/libx.a` libraries.
`-L`*`dir`*	Links the libraries in *`dir`* before the libraries in the default search path.
`-minshared`	Indicates that the result of the current compilation is going into an executable file that will make minimal use of shared libraries. This option is equivalent to `-exec -Bprotected`.
`-mt`	Enables multi-threading capability with the need to set flags.
`-n`	Generates shareable code.
`-N`	Generates unshareable code.
`-o` *`outfile`*	Places object modules in *`outfile`* file.
`-O`	Optimizes at level 2.

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

Option	Description
`-p`	Inserts information required by the `prof` profiler in the object file.
`-P`	Performs preprocessing only with output to the corresponding `.`*`i`* file.
`-q`	Marks the executable as demand loadable.
`-Q`	Marks the executable as not being demand loadable.
`-r`	Retains relocation information in the output file for subsequent relinking.
`-s`	Strips the symbol table from the executable file.
`-S`	Generates an assembly language source file.
`-t` *`x,name`*	Substitutes or inserts subprocess *`x`* with *`name`*.
`-U`*`name`*	Undefines *`name`* in the preprocessor.
`-v`	Enables verbose mode.
`-V`	Causes subprocesses to print version information to `stderr`.
`-w`	Suppresses warning messages.
`-W`*x*, *`arg1 [,arg2,..,argn]`*	Passes the arguments *`arg1`* through *`argn`* to the subprocess *`x`*.
`-Wc,-ansi_for_scope [on|off]`	Enables or disables the standard scoping rules for `init` declarations in `for` statements.
`-Wc,-koenig_lookup,[on|off]`	Enables or disables standard argument-dependent lookup rules.
`-Y`	Enables Native Language Support (NLS).
`-z`	Disallows runtime dereferencing of null pointers.
`-Z`	Allows dereferencing of null pointers at runtime.

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

Option	Description
`+cond_rodata`	Allows more data to be placed in a read-only section.
`+d`	Prevents expansion of inline functions.
`+DD32`	Generates ILP32 code and is the default.
`+DD64`	Generates LP64 code.
`+DS`*`model`*	Performs instruction scheduling for a specific implementation of Itanium®-based architecture.
`+DO`*`osname`*	Sets the target operating system for the compiler.
`+dryrun`	Generates subprocesses information for a given command line without running the subprocesses.
`+e`	Enables the following HP value added features while compiling in ANSI C mode: `sized enum`, `long long`, long pointers, compiler supplied defaults for missing arguments to intrinsic calls, and `$` in identifier HP C extensions.
`+ES[no]lit`	Places [does not place] string literals and const-qualified variables that do not require load-time or runtime initialization in the read-only data section.
`+FP`*`flags`*	Controls floating-point traps.
`+help`	Launches a web browser displaying an html version of the HP C/HP-UX Online Help.
`+ild`	Specifies incremental linking.
`+ildrelink`	Performs an initial incremental link, regardless of the output load module.
`+inline_level num`	Controls how C++ inlining hints influence HP aC++.
`+inst_compiletime`	Causes the compiler to use the compile time (CTTI) instantiation.

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

Option	Description
`+inst_directed`	Indicates to the compiler that no templates are to be instantiated (except explicit instantiations).
`+inst_implicit_include`	Specifies that the compiler use a process similar to the cfront source rule for locating template definition files.
`+inst_include_suffixes`	Specifies the file name extensions that the compiler uses to locate definition files.
`+legacy_cpp`	Enables the use of the HP-UX 11.20 ANSI C preprocessor.
`+legacy_v5`	Enables the use of the A.05.* compiler.
`+M`	Provides ANSI migration warnings that explain the differences between code compiled with `-Ac` and `-Aa`.
`+M1`	Provides platform migration warnings for features that may not be supported in future releases.
`+M2`	Provides migration warnings for transitioning code from the ILP32 to the LP64 data model.
`+m[d]`	Directs a list of quote enclosed header files to `stdout`.
`+M[d]`	Directs a list of both quote enclosed header files and angle bracket enclosed header files to `stdout`.
`+noeh`	Disables exception handling.
`+nostl`	Eliminates references to standard header files and libraries to allow developers full control over header files and libraries in compilation and linking of their applications.
`+[no]nrv`	Enables [disables] named value return (NRV) optimization.

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

Option	Description
`+o`	Prints hexadecimal code offsets in the source code listing.
`+O[no]cross_region_addressing`	Enables [disables] the use of cross-region addressing.
`+O[no]cxlimitedrange`	Enables [disables] the use of the usual formulas for complex arithmetic.
`+O[no]datalayout`	Enables [disables] profile-driven layout of global and static data items to improve cache memory utilization.
`+O[no]dataprefetch`	When `+Odataprefetch` is enabled, the optimizer inserts instructions within innermost loops to explicitly prefetch data from memory into the data cache.
`+O[no]failsafe`	Enables [disables] failsafe optimization.
`+O[no]fenvacess`	Provides a means to inform the compiler when a program might access the floating point environment to test flags or run under non-default modes.
`+O[no]fltacc`	Disables [enables] floating-point optimizations that can result in numerical differences.
`+O[no]info`	Displays information about the optimization process.
`+O[no]initcheck`	Initializes [does not initialize] local and non-static variables.
`+O[no]inline`	Allows [does not allow] inlining of funtions by the optimizer.
`+O[no]libmerrno`	Enables [disables] support for `errno` in `libm` funtions.
`+O[no]limit`	Enables [disables] optimizations that significantly affect compile time or memory consumption.

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

Option	Description
`+O[no]loop_transform`	Transforms [does not transform] eligible loops for improved cache and performance.
`+O[no]loop_unroll`	Enables [disables] loop unrolling.
`+O[no]loop_unroll_jam`	Enables [disables] loop unrolling and jamming.
`+O[no]openmp`	Honors [silently ignores] OpenMP directives.
`+O[no]parminit`	Enables [disables] automatic initialization to non-NaT function parameters at call sites.
`+O[no]preserved_fpregs`	Specifies whether the compiler is allowed [not allowed] to make use of the preserved subset of the floating point register file.
`+O[no]procelim`	Enables [disables] the elimination of dead procedure code and unreferenced data.
`+O[no]promote_indirect_calls`	Uses profile data from profile-based optimization and other information to determine the most likely target of indirect calls and promotes them to direct calls.
`+O[no]ptrs_to_globals`	Tells the optimizer whether global variables are accessed [not accessed] through pointers.
`+O[no]recovery`	Generates [does not generate] recovery code for control speculation.
`+O[no]report`	Displays [does not display] optimization reports.
`+O[no]signedopinters`	Treats [does not treat] pointers in Boolean comparisons as signed quantities.
`+O[no]sumreduction`	Enables [disables] sum reduction optimization.
`+O[no]size`	While most optimizations reduce code size, the `+Osize` option suppresses those few optimizations that significantly increase code size. The `+Onosize` option enables code-expanding optimizations.

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

Option	Description
`+O[no]store_ordering`	Preserves [does not preserve] the original program order for stores to memory that is visible to multiple threads.
`+O[no]volatile`	Treats all global variables as [not] volatile.
`+O[no]whole_program_mode`	Enables [disables] assertion for files compiled with this option.
`+O[0|1|2|3|4]`	Specifies the level of optimization.
`+Ofast`	Selects a combination of optimization options for optimum execution speed and reasonable build times.
`+Ofaster`	Equivalent to `+Ofast` with an increased optimization level.
`+Ofrequently_called`	Calls a list of functions that are frequently called. This option overrides any information in a profile database.
`+Oinlinebudget`	Controls the compile time budget for the inliner.
`+Ointeger_overflow`	Ensures that runtime integer arithmetic expressions that arise in certain contexts do not overflow.
`+Olevel`	Lowers optimization to a specified level for one or more functions.
`+Olit`	Places data items that do not require load-time or runtime initialization in a read-only data section.
`+Oprefetch_latency`	Applies to loops for which the compiler generates data prefetch instructions.
`+Orarely_called`	Calls a list of functions that are rarely called. This option overrides any information in a profile database.

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

Option	Description
`+Oshortdata`	Places objects in the short data area; and references to such data assume it resides in the short data area
`+Otype_safety`	Controls type-based aliasing assumptions.
`+Ounroll_factor`	Applies the unroll factor to all loops in the current translation unit.
`+Oprofile=collect`	Prepares the object code for profile-based optimization data collection.
`+Oprofile=use`	Performs profile-based optimization.
`+[no]objdebug`	When used with `-g`, `+objdebug` leaves debug information in the object files instead of copying it to the executable file. The object files must be accessible to the HP WDB debugger when debugging.
`+p`	Disallows all anachronistic constructs.
`+profilebucketsize`	Enables support for `prof` and `gprof` when building an executable, but not a shared library.
`+sb`	Make bitfields signed by default in both 32-bit and 64-bit modes.
`+[no]srcpos`	Controls generation of source position information for HP Caliper.
`+time`	Generates timing information for compiler subprocesses.
`+tls`	Specifies whether references to thread local data items are to be performed according to the mode.
`+tru64`	Causes return types of unprototyped functions to be treated as long, instead of int, matching Tru64 behavior.
`+ub`	Specifies unqualified `char`, `short`, `int`, `long`, and `long long` bit fields as unsigned.

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

Option	Description
`+uc`	Makes unqualified `char` data types `unsigned`.
`+w`	Warns about all questionable constructs and gives pedantic warnings.
`+W` *arg1*[,*arg2*,...arg*n*]	Suppresses the specified warnings.
`+We` *n1*[,*n2*,...*nN*]	Changes the specified warnings to errors.
`+wn`	Specifies the level of warning messages. Warns about all questionable constructs and gives pedantic warnings.
`+Ww` *n1*[,*n2*,...*nN*]	Enables the specified warnings, assuming all other warnings are suppressed with -w or +w3.
`+z`	Generates shared library object code (same as `+Z` in 64-bit mode).
`+Z`	Generates shared library object code with a large data linkage table (long-form PIC).

Examples of Compiler Commands

- `cc -Aa prog.c`

  requests a strict ANSI C compilation of `prog.c`.

- `cc -tp,/users/devel/cpp prog.c`

  uses `/users/devel/cpp` as the pathname for the preprocessing phase.

- `cc -tpca,/users/devel/x prog.s`

  uses `/users/devel/x/cpp` for cpp, `/users/devel/x/ctcom` for `ctcom`, and `/users/devel/x/as` for `as`; the assembly file `prog.s` is processed by the specified assembler.

- `cc -Aa prog.c procedure.o -o prog`

  compiles and links the file `prog.c`, creating the executable program file `prog`. The compiler produces `prog.o`. The linker `ld(1)` links `prog.o` and `procedure.o` with all of the HP C startup routines and the library routines from the HP C library `/usr/lib/hpux32/libc.so`.

- `cc prog.c -co /users/my/prog.o`

  compiles the source file `prog.c` and places the object file `prog.o` in `/users/my/prog.o`.

- `cc -Wp,-H150000 p1.c p2.c p3.c -o p +legacy_cpp`

  compiles the source files in the option `-H150000` to the preprocessor `cpp` to increase the define table size from the default.

- `cc -Wl,-vt *.c -o vmh`

  compiles all files in the working directory ending with `.c`, passes the `-vt` option to the linker, and causes the resulting program file to be named `vmh`.

- `cc -vg prog.c`

  compiles `prog.c`, adds debug information, and displays the steps in the compilation process.

- `cc -S prog.c`

  compiles the file `prog.c` into an assembly output file called `prog.s`.

- `cc -OAa prog.c`

  compiles `prog.c` in ANSI mode and requests level 2 optimization.

- `cc +O1 prog.c`

  compiles `prog.c` and requests level 1 optimization.

- `cc +w1 prog.c -c`

  compiles `prog.c` with low-level warnings emitted and suppresses linking.

- `cc -D BUFFER_SIZE=1024 prog.c`

  passes the option `-D BUFFER_SIZE=1024` to the preprocessor, setting the value of the macro for the compilation of `prog.c`.

- `cc prog.c -lm`

  compiles `prog.c` requests the linker to link the library `/usr/lib/hpux32/libm.so` with the object file `prog.o` to create the executable `a.out`.

- `cc -L/users/devel/lib prog.c -lme`

  compiles `prog.c` and causes the linker to search the directory `/users/devel/lib` for the library `libme.a`, before searching in `/usr/lib/hpux32`.

- `cc +DD32 +Z -c prog.c`

  compiles `prog.c` for use in a large shared library in 32-bit mode.

- `cc +DD64 +Z -c prog.c`

  compiles `prog.c` for use in a large shared library or executable in 64-bit mode.

- `cc +DD32 +z -c prog.c`

  compiles `prog.c` in 32-bit mode for use in a shared library with less than 16K of symbols.

- `cc +DD64 prog.c`

  compiles `prog.c` in 64-bit mode.

  Refer to the discussion on optimization options in the C online help or the *HP C/HP-UX Programmer's Guide* for more information.

Environment Variables

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

- CCOPTS Environment Variable
- CCROOTDIR Environment Variable
- TMPDIR Environment Variable
- aCC_MAXERR Environment Variable

CCOPTS Environment Variable

You can pass arguments to the compiler using the `CCOPTS` environment variable or include them on the command line. The `CCOPTS` environment variable provides a convenient way for establishing default values for `cc` command line options. It also provides a way for overriding `cc` command line options.

The syntax for the `CCOPTS` environment variable in C shell notation is:

`export CCOPTS="options | options" (ksh/sh` notation)

`setenv CCOPTS "options | options" (csh` notation)

The compiler places the arguments that appear before the vertical bar in front of the command line arguments to `cc`. It then places the second group of arguments after any command line arguments to `cc`.

Options that appear after the vertical bar in the `CCOPTS` variable override and take precedence over options supplied on the `cc` command line.

If the vertical bar is omitted, the compiler gets the value of `CCOPTS` and places its contents before any arguments on the command line.

For example, the following in C shell notation

```
setenv CCOPTS -v
cc -g prog.c
```

is equivalent to

```
cc -v -g prog.c
```

For example, the following in C shell notation

```
setenv CCOPTS "-v | +O1"
cc +O2 prog.c
```

is equivalent to

```
cc -v +O2 prog.c +O1
```

In the above example, level 1 optimization is performed, since the `+O1` argument appearing after the vertical bar in `CCOPTS` takes precedence over the `cc` command line arguments.

CCROOTDIR Environment Variable

The `CCROOTDIR` environment variable causes `aCC` to invoke all subprocesses from an alternate `aCC` directory, rather than from their default directory. The default aCC root directory is `/opt/aCC`.

Syntax:

```
export CCROOTDIR=directory   ksh/sh notation

setenv CCROOTDIR directory   csh notation
```

`directory` is an aCC root directory where you want the HP aC++ driver to search for subprocesses.

Example:

```
export CCROOTDIR=/mnt/CXX2.1
```

In this example, HP aC++ searches the directories under `/mnt/CXX2.1` (`/mnt/CXX2.1/bin` and `/mnt/CXX2.1/lbin`) for subprocesses rather than their respective default directories.

TMPDIR Environment Variable

Another environment variable, `TMPDIR,` allows you to change the location of temporary files that the compiler creates and uses. The directory specified in `TMPDIR` replaces `/var/tmp` as the default directory for temporary files. The syntax for `TMPDIR` in C shell notation is:

`export TMPDIR=directory` (`ksh/sh` notation)

`setenv TMPDIR directory` (`csh` notation) `directory` is the name of an HP-UX directory where you want HP C to put temporary files during compilation.

aCC_MAXERR Environment Variable

This release of HP C compiler provides support to specify the maximum number of errors emitted before the compilation aborts. In the earlier versions, the compiler stopped, if it recognized more than 99 errors while compiling. With this release, the maximum number of errors that the compiler emits can be tuned as required by setting the `aCC_MAXERR` environment variable.

The syntax for `aCC_MAXERR` in C shell notation is:

```
export aCC_MAXERR=errors
```

where *`errors`* are the maximum number of errors set.

SDK/XDK

SDK/XDK, helps you in selecting the components, headerfiles, and libraries installed in alternate locations. To enable this support in your compiler, you need to set either one or both of the following environment variables:

- `SDKROOT`
- `TARGETROOT`

Setting SDKROOT Environment Variable

The SDKROOT environment variable is used as a prefix for all references to tool set components. This environment variables is set by the user while using a non-native development kit or toolset installed at an alternative location. Some of the toolset components are compiler drivers, compiler applications, preprocessor, linker, and object file tools. If the HP C compiler has its tool set installed in `/opt/xdk-ia/directory` then the command:

```
export SDKROOT=/opt/xdk-ia
```

will prefix all references to the HP C compiler tool set components with `/opt/xdk-ia`. The following details the default tool set components location as specified in the above command and its earlier location before the execution of the command:

Table 9-2 Native and Alternate Toolset Location

Native Location	Alternate Toolset Location
`/opt/ansic/bin/cc`	`/opt/xdk-ia/opt/ansic/bin/cc`

Table 9-2 Native and Alternate Toolset Location (Continued)

Native Location	Alternate Toolset Location
`/opt/ansic/lbin/ecom` (for A.06.*)	`/opt/xdk-ia/opt/ansic/lbin/ecom` (for A.06.*)
`/opt/ansic/lbin/ctcom` (for A.05.*)	`/opt/xdk-ia/opt/ansic/lbin/ctcom` (for A.05.*)
`/opt/langtools/lbin/cpp.ansi`	`/opt/xdk-ia/opt/langtools/lbin/cpp.ansi`
`/opt/langtools/lbin/u2comp`	`/opt/xdk-ia/opt/langtools/lbin/u2comp`
`/opt/langtools/lbin/be`	`/opt/xdk-ia/opt/langtools/lbin/be`

Invoking the toolset components will actually result in the invocation of tool set components from the alternate location as specified above.

Setting TARGETROOT Environment Variable

The TARGETROOT environment variable is used as a prefix for all references to target set components. This environment variable is set by the user when using a non-native development kit. Some of the target set components are header files, archive libraries, and shared libraries. If the HP C compiler has its large tool set installed in `/opt/xdk-ia/` directory, the command:

```
export TARGETROOT=opt/xdk-ia
```

will prefix all references to the target tool set components with `/opt/xdk-ia`. The following details the default location of the tool set components as specified in the above command and its earlier location before the execution of the command:

Table 9-3 Native and Alternate Toolset Location

Native location	Alternate Toolset Location
`/usr/include`	`/opt/xdk-ia/usr/include`
`/usr/lib`	`/opt/xdk-ia/usr/lib`

NOTE Options like `-l` or `-L` on the command line will override `$TARGETROOT` prefixing.

Pragmas

A `#pragma` directive is an instruction to the compiler. Put pragmas in your C source code where you want them to take effect, but do not use them within a function. A pragma has effect from the point at which it is included to the end of the translation unit (or until another pragma changes its status).

This section introduces the following groups of HP C compiler directives:

- Initialization and Termination Pragmas
- Copyright Notice and Identification Pragmas
- Data Alignment Pragmas
- Optimization Pragmas
- FastCall Pragmas
- Gather/Scatter Prefetch Pragma

Initialization and Termination Pragmas

This section describes the `INIT` and `FINI` pragmas. These allow the user to set up functions which are called when a load module (a shared library or executable) is loaded (initializer) or unloaded (terminator). For example, when a program begins execution, its initializers get called before any other user code gets called. This allows some set up work to take place. In addition, when the user's program ends, the terminators can do some clean up. When a shared library is loaded or unloaded with the `shl_load` or `dlopen` API, its initializers and terminators are also executed at the appropriate time.

INIT Pragma

`#pragma INIT "`*`string`*`"`

Use the compiler pragma `INIT` to specify an initialization function. The functions take no arguments and return nothing. The function specified by the `INIT` pragma is called before the program starts or when a shared library is loaded.

For example:

```
#pragma INIT "my_init"
void my_init() {
   ...do some initializations ...
}
```

FINI Pragma

```
#pragma FINI “string”
```

Use the compiler pragma `FINI` to specify a termination function. The function specified by the `FINI` pragma is called after the C program terminates by either calling the libc `exit()` function, returning from the `main` or `_start` functions, or when the shared library which contains the `FINI` is unloaded from memory. Like the function called by the `INIT` pragma, the termination function takes no arguments and returns nothing.

For example:

```
# pragma FINI “my-fini”
void my_fini() {
   ...do some clean up ...
}
```

Copyright Notice and Identification Pragmas

The following pragmas can be used to insert strings in code.

COPYRIGHT Pragma

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

Places a copyright notice in the object file, using the “*string*” argument and the date specified using `COPYRIGHT_DATE`. If no date has been specified using `#pragma COPYRIGHT_DATE`, the current year is used. For example, assuming the year is 2004, the directive `#pragma COPYRIGHT "Acme Software"` places the following string in the object code:

```
(C) Copyright Acme Software, 2004. All rights reserved. No part
of this program may be photocopied, reproduced, or transmitted
without prior written consent of Acme Software.
```

COPYRIGHT_DATE Pragma

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

Specifies a date string to be used in a copyright notice appearing in an object module.

LOCALITY Pragma

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

Specifies a name to be associated with the code written to a relocatable object module. All code following the `LOCALITY` pragma is associated with the name specified in *string*. The smallest scope of a unique `LOCALITY` pragma is a function.

For example, the following command builds the name `$CODE&MINE$`:

```
#pragma locality "mine"
```

Code that is not headed by a `LOCALITY` pragma is associated with the name `$CODE$`. An empty "*string*" causes the code name to revert to the default name of `$CODE$`.

VERSIONID Pragma

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

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

Data Alignment Pragmas

This section discusses the data alignment pragmas `HP_ALIGN` and PACK and their various arguments available on the HP-UX systems, to control alignment across platforms. In the following discussion, a word represents a 32-bit data structure. Refer to *HP aC++/HP C Programmer's Guide* for detailed information on the `HP_ALIGN` and PACK pragmas.

ALIGN Pragma

```
#pragma align N
```

`N` is a number raised to the power of 2.

HP aC++ supports user specified alignment for global data. The pragma takes effect on next declaration. If the align pragma declaration is not in the global scope or if it is not a data declaration, the compiler displays a warning message. If the specified alignment is lesser than the original alignment of data, a warning message is displayed, and the pragma is ignored.

```
#pragma align 2

char c;              // "c" is at least aligned on 2 byte boundary
#pragma align 64
   int i, a[10];     // "i" and array "a" are at least aligned 64 byte boundary.
                       // the size of "a" is still 10*sizeof(int)
```

PACK Pragma

```
#pragma PACK n
```

The `PACK` pragma is a simple, intuitive way for users to specify alignment. In the syntax, *n* is the byte boundary on which members of structs and unions should be aligned, and can be 1, 2, 4, 8, or 16.

The PACK pragma is not intended to be an "extension" of the HP_ALIGN pragma. It is, instead, a simple and highly portable way of controlling the alignment of aggregates. It has some significant differences with the HP_ALIGN pragma, including uniform bitfield alignment, uniform struct and union alignment, and the lack of PUSH and POP functionality.

For complete details on the use of HP_ALIGN and PACK pragmas, refer to Chapter 2, "Storage and Alignment Comparisons," in the *HP C/HP-UX Programmer's Guide*.

UNALIGN Pragma

```
#pragma unalign [1|2|4|8|16]
```

```
typedef T1 T2;
```

T1 and T2 have the same size and layout, but with specified alignment requirements.

HP C supports misaligned data access using the unalign pragma. The unalign pragma can be applied on typedef to define a type with special alignment. The unalign pragma takes effect only on next declaration.

If the unalign pragma declaration is not in the global scope or if it is not a typedef, compiler displays a warning message. If the specified alignment is greater than the original alignment of the declaration, then an error message is displayed, and the pragma is ignored. For example,

```
#pragma unalign 1
typedef int ua_int;            // ua_int is of int type
                               // with 1 byte alignment

typedef ua_int *ua_intPtr;     // this typedef is not affected
                               // affected by the above unalign
                               // pragma. it defines a pointer
                               // type which points to 1 byte
                                 // aligned int
```

The interaction between pack and unalign pragmas is as follows:

```
#pragma pack 1
struct S {
   char c;
   int i;
};

#pragma pack 0

S s;
ua_int *ua_ip = &s.i;          // ua_ip points to 1 byte aligned int
*ua_ip = 2;                     // mis-aligned access to 1 byte aligned int
```

NOTE The `HP_ALIGN` pragma, which is supported by HP ANSI C compiler, is not supported by HP aC++. The `pack` and `unalign` pragmas can replace most of the `HP_ALIGN` functionality.

Optimization Pragmas

For additional information on the following optimization pragmas refer to *HP aC++/HP C Programmer's Guide*.

FLOAT_TRAPS_ON Pragma

```
#pragma FLOAT_TRAPS_ON { functionname,...functionname }
```

```
#pragma FLOAT_TRAPS_ON {_ALL }
```

The `FLOAT_TRAPS_ON` pragma informs the compiler that you may have enabled floating-point trap handling. When the compiler is so informed, it will not perform loop invariant code motion (LICM) on floating-point operations in the functions named in the pragma. This pragma is required for proper code generation when floating-point traps are enabled and the code is optimized.

The `_ALL` parameter specifies that loop invariant code motion should be disabled for all functions within the compilation unit.

[NO]INLINE Pragma

#pragma INLINE [*functionname*$_1$,...,*functionname*$_n$]

#pragma NOINLINE [*functionname*$_1$,...,*functionname*$_n$]

Enables (or disables) inlining of functions. If particular functions are specified with the pragma, they are enabled (or disabled) for inlining. If no functions are specified with the pragmas, all functions are enabled (or disabled) for inlining. Refer to the HP *C/HP-UX Programmer's Guide* for details and examples.

NO_SIDE_EFFECTS Pragma

#pragma NO_SIDE_EFFECTS *functionname*$_1$,...,*functionname*$_n$

States that *functionname* and all the functions that *functionname* calls will not modify any of a program's local or global variables. This pragma provides additional information to the optimizer which results in more efficient code. See the *HP C/HP-UX Programmer's Guide* for further information.

FastCall Pragmas

The compiler directives described in this section are designed to speed up shared library calls.

HP_DEFINED_EXTERNAL Pragma

```
#pragma HP_DEFINED_EXTERNAL sym1, sym2, ...
```

The externally defined symbol pragma specifies that the designated symbols are imported from another load module (program file or shared library). Note that this pragma currently works in 32-bit mode only. For 64-bit mode, use the option `+Oextern`.

HP_LONG_RETURN Pragma

```
#pragma HP_LONG_RETURN func1, func2, ...
```

The long return sequence pragma specifies that the named procedures can return directly across a space, without needing to return through an export stub. The main goal of this pragma is to eliminate export stubs, and better enable inlining of import stubs and `$$dyncall` functionality for indirect calls.

HP_NO_RELOCATION Pragma

```
#pragma HP_NO_RELOCATION func1, func2, ...
```

The no parameter/return relocation pragma is used to suppress unnecessary floating point argument relocation. When used, the burden is on the caller and callee to have their argument and their return values agree in which register files they are passed.

Gather/Scatter Prefetch Pragma

This release supports pragmas for prefetching the cache lines specified in the pragma.

Syntax

```
#pragma prefetch <argument>
```

The behavior of this pragma is similar to the `HP_OPT_DATA` pragma which prefetches the data specified. But the prefetch pragma can access cache lines that are accessed via a vector of indices.

`<argument>` can have only one argument per pragma and it must me an array element. For example, `a[i]`.

The compiler generates instructions to prefetch the cache lines starting from the address of an argument. The values prefetched must be valid values. Reading off the end of an array may result in undefined behavior during runtime.

Example:

The function below prefetches `ia` and `b`, but not `a[ia[i]]` when compiled with `+O2 +Odataprefetch +DA2.0` (or `+DA2.0W`).

```
testprefc2(int n, double *a, int *ia, double *b)

{
  for (int i=0; i<n, i++)  {
       b[i]=a[ia[i]];
  }
}
Recording this routine as

#define USER_SPECIFIED  30

testprefc2(int n, double *a, int *ia, double *b)
{
  int dist=(int)USER_SPECIFIED;
  int nend=max(0,n_dist); /* so as not to read past the end of is */
  for(i=0;i<nend;i++) /* original loop is for (i=0;i<n;i++) */
  {
    #pragma prefetch ia[i+4*dist]
    #pragma prefetch a[ia[i+dist]]
    b[i]=a[ia[i]];
  }
/* finish up last part with no prefetching  */

  for (int i=nend;i<n;i++)
  b[i]=a[ia[i]];
}
```

Running HP C Programs

After a program is successfully linked, it is in executable form. To run the program, enter the executable filename (either `a.out` or the name following the `-o` option).

10 HP C/HP-UX Implementation Topics

This chapter describes the following topics that are specific to programming in C on HP-UX systems:

Data Types

Data types are implemented in HP C/HP-UX as follows:

- The `char` type is signed.
- All types can have the `register` storage class, although it is only honored for scalar types. Ten register declarations per function are honored.
- The signed integer types are represented internally using twos complement form.
- Structures and unions start and end on an alignment boundary which is that of their most restrictive member.
- The `long long` data type cannot be used to declare an array's size.
- The `long long` data type is available only under `-Ac`, `-Aa +e`, and `-Ae` compilation modes.

Table 10-1 lists the sizes and ranges of different HP C/HP-UX data types.

Table 10-1 HP C/HP-UX Data Types

Type	Bits	Bytes	Low Bound	High Bound	Comments
`char`	8	1	-128	127	Character
`signed char`	8	1	-128	127	Signed integer
`unsigned char`	8	1	0	255	Unsigned integer
`short`	16	2	-32,768	32,767	Signed integer
`unsigned short`	16	2	0	65,535	Unsigned integer
`int`	32	4	-2,147,483,648	2,147,483,647	Signed integer
`unsigned int`	32	4	0	4,294,967,295	Unsigned integer

Table 10-1 HP C/HP-UX Data Types (Continued)

Type	Bits	Bytes	Low Bound	High Bound	Comments
`long` (ILP32)	32	4	-2,147,483,648	2,147,483,647	Signed integer
`long` (LP64)	64	8	-2^{63}	2^{63} -1	Signed integer
`long long`	64	8	-2^{63}	2^{63} -1	Signed integer
`unsigned long` (ILP32)	32	4	0	4,294,967,295	Unsigned integer
`unsigned long` (LP64)	64	8	0	2^{64} -1	Unsigned integer
`unsigned long long`	64	8	0	2^{64} -1	Unsigned integer
`float`	32	4	See (a) below.	See (b) below.	Floating-point
`double`	64	8	See (c) below.	See (d) below.	Floating-point
`long double`	128	16	See (e) below.	See (f) below.	Floating-point
`enum`	32	4	-2,147,483,648	2,147,483,647	Signed integer

Comments

In the following comments, the low bounds of `float`, `double`, and `long double` data types are given in their *`normalized`* and *`denormalized`* forms. Normalized and denormalized refer to the way data is stored. Normalized numbers are represented with a greater degree of accuracy than denormalized numbers. Denormalized numbers are very small numbers represented with fewer significant bits than normalized numbers.

a. Least normalized: 1.17549435E-38F
Least denormalized: 1.4012985E-45F

b.	3.40282347E+38F
c.	Least normalized: 2.2250738585072014E-308 Least denormalized: 4.9406564584124654E-324
d.	1.7976931348623157E+308
e.	Least normalized: 3.3621031431120935062626778173217526026E-4932L Least denormalized: 6.4751751194380251109244389582276465525E-4966L
e.	1.1897314953572317650857593266280070162E+4932L

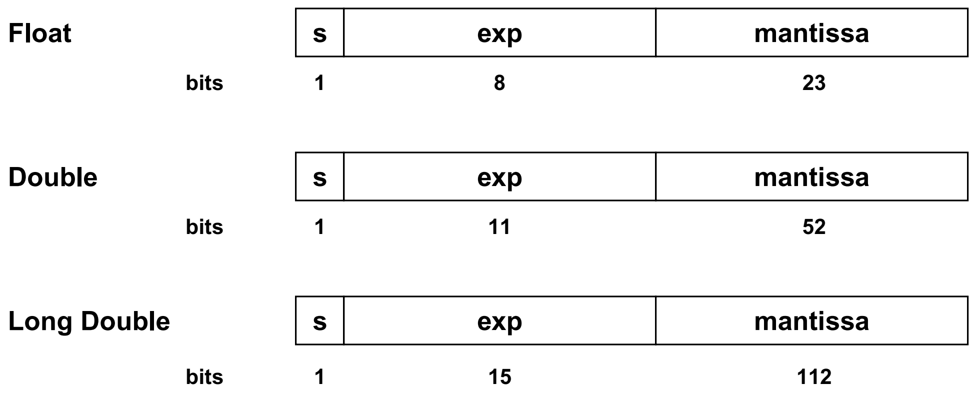
Bit-Fields

- Bit-fields in structures are packed from left to right (high-order to low-order).
- The high order bit position of a “plain” integer bit-field is treated as a sign bit.
- Bit-fields of types `char`, `short`, `long`, `long long`, and `enum` are allowed.
- The maximum size of a bit-field is 64 bits.
- If a bit-field is too large to fit in the current word, it is moved to the next word.
- The range of values in an integer bit-field are:
  - -2,147,483,648 to 2,147,483,647 for 32-bit signed quantities
  - 0 to 4,294,967,295 for 32-bit unsigned quantities
  - -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 for 64-bit signed quantities
  - 0 to 18,446,744,073,709,551,615 for 64-bit unsigned quantities
- Bit-fields in unions are allowed only in ANSI mode.

IEEE Floating-Point Format

The internal representation of floating-point numbers conforms to the IEEE floating-point standard, ANSI/IEEE 754-1985, as shown in Figure 10-1.

Figure 10-1 Internal Representation of Floating-Point Numbers



The `s` field contains the sign of the number. The `exp` field contains the biased exponent (`exp = E + bias`, where `E` is the real exponent) of the number. The values of `bias` and the maximum and minimum values of the unbiased exponent appear in the following table:

Table 10-2 Floating-point Bias, and Unbiased Exponents Values

	float	double	long double
bias	+127	+1023	+16383
Emax	+127	+1023	+16383
Emin	-126	-1022	-16382

E_{min}-1 is used to encode 0 and denormalized numbers.

E_{max}+1 is used to encode infinities and NaNs.

NaNs are binary floating-point numbers that have all ones in the exponent and a nonzero fraction. NaN is a term used for a binary floating-point number that has no value (that is, “Not A Number”).

If `E` is within the range

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

the mantissa field contains the number in a normalized form, preceded by an implicit 1 and binary point.

In accordance with the IEEE standard, floating-point operations are performed with traps not enabled, and the result of such an operation is that defined by the standard. This means, for example, that dividing a positive finite number by zero will yield positive infinity, and no trap will occur. Dividing zero by zero or infinity by infinity will yield a NaN, again with no trap. For a discussion of infinity arithmetic and operations with NaNs, in the context of the IEEE standard, see the *HP Precision Architecture and Instruction Set Reference Manual* (HP part number 09740-90014).

For detailed information about floating-point arithmetic on HP-UX, how HP-UX implements the IEEE standard, and the HP-UX math libraries, see the *HP-UX Floating Point Guide*.

Note that infinities and NaNs propagate through a sequence of operations. For example, adding any finite number to infinity will yield infinity. An operation on a NaN will yield a NaN. This means that you may be able to perform a sequence of calculations and then check just the final result for infinity or NaN.

The HP-UX math library provides routines for determining the class of a floating point number. For example, you can determine if a number is infinity or `NAN`. See the *HP-UX Reference* for descriptions of the functions `fpclassify`, `fpclassifyf`, `isinf`, and `isnan`.

Lexical Elements

- Identifiers: 255 characters are significant in internal and external names.
- Character Constants: Any character constant of more than one character produces a warning. The value of an integral character constant containing more than one character is computed by concatenating the 8-bit ASCII code values of the characters, with the leftmost character being the most significant. For example, the character constant `'AB'` has the value `256*'A'+'B' = 256*65+66 = 16706`. Only the rightmost four characters participate in the computation.
- The case of alphabetic characters is always significant in external names.
- The execution character set and the source character set are both ASCII.
- Nonprinting characters in character constants and string literals must be represented as escape sequences.

Structures and Unions

Structure or union references that are not fully qualified (see example below) are flagged with an error by the compiler.

```
struct{
       int j;
       struct {int i;}in;
      } out;
      out.i=3;
```

The correct statement for the example above is `out.in.i = 3;`.

Type Mismatches in External Names

It is illegal to declare two externally visible identifiers of different types with the same name in separately compiled translation units. The linker might not diagnose such a mismatch.

Expressions

The value of an expression that overflows or underflows is undefined, except when the operands are unsigned.

Pointers

- Pointers to functions should not be compared using relational operators because the pointers represent external function labels and not actual addresses.
- Dereferencing a pointer that contains an invalid value results in a trap if the address references protected memory or if the address is not properly aligned for the object being referenced.
- A declaration of a pointer to an undefined structure tag is allowed, and the tag need not be defined in the source module unless the pointer is used in an expression.

Maximum Number of Dimensions of an Array

Arrays can have up to 252 dimensions.

Scope of extern Declarations

Identifiers for objects and functions declared within a block and with the storage class `extern` have the same linkage as any visible declaration with file scope. If there is no visible declaration with file scope, the identifier has external linkage, and the definition remains visible until the end of the translation unit.

However, because this is an extension to ANSI C, a warning will be issued on subsequent uses of the identifier if the absence of this extended visibility could cause a change in behavior on a port to another conforming implementation.

Conversions Between Floats, Doubles, and Long Doubles

- When a `long double` is converted to a `double` or `float`, or when a `double` is converted to a `float,` the original value is rounded to the nearest representable value as the new type. If the original value is equally close to two distinct representable values, then the value chosen is the one with the least significant bit equal to zero.
- Conversions between floating-point types involve a change in the exponent, as well as the mantissa. It is possible for such a conversion to overflow.

Statements

- The types of `switch` expressions and their associated `case` label constants do not need to match. Integral types can be mixed.
- All expressions of integral types are allowed in `switch` statements.

Preprocessor

- The maximum nesting depth of `#include` files is 35.
- For include files that are enclosed in double quotes and do not begin with a `/`, the preprocessor will first search the current directory, then the directory named in the `-I` option, and finally, in the standard include directory `/usr/include`.
- For include files that are enclosed in < and > signs, the search path begins with the directory named in the `-I` option and is completed in the standard include directory, `/usr/include`. The current directory is not searched.

Library Functions and Header Files

This section describes the implementation of library functions in HP C/HP-UX. For complete information about library functions on HP C/HP-UX, see the `HP-UX Reference` manual and *HP-UX Linker and Libraries Online User Guide*.

The Math Library

When using any of the mathematical functions in the `<math.h>` header, you must include the `-lm` flag on the `cc` or `ld` command when linking. This will cause the linker to link in the appropriate math library.

Other Library Functions

- **longjmp**: Because HP C/HP-UX can place automatic variables in registers, you cannot rely on their values if they are changed between the `setjmp` and `longjmp` functions.
- **setjmp**: There are no restrictions on when calls to `setjmp` can be made.

NOTE `+Olibcalls` transforming `setjmp` to `_setjmp` and `longjmp` to `_longjmp` is not supported in 11.x of HP C compilers.

The varargs Macros

The varargs macros allow accessing arguments of functions where the number and types of the arguments can vary from call to call.

NOTE The <varargs.h> header has been superseded by the standard header <stdarg.h>, which provides all the functionality of the varargs macros. The <varargs.h> header is retained for compatibility with pre-ANSI compilers and earlier releases of HP C/HP-UX.

To use varargs, a program must include the header <varargs.h>. A function that expects a variable number of arguments must declare the first variable argument as va_alist in the function declaration. The macro va_dcl must be used in the parameter declaration.

A local variable should be declared of type va_list. This variable is used to point to the next argument in the variable argument list.

The va_start macro is used to initialize the argument pointer to the initial variable argument.

Each variable argument is accessed by calling the va_arg macro. This macro returns the value of the next argument, assuming it is of the specified type, and updates the argument pointer to point to the next argument.

The va_end macro is provided for consistency with other implementations; it performs no function on HP systems. The following example demonstrates the use of the <varargs.h> header:

Example

```
#include <varargs.h>
#include <stdio.h>

enum arglisttype {NO_VAR_LIST, VAR_LIST_PRESENT};
enum argtype {END_OF_LIST, CHAR, DOUB, INT, PINT};

int foo (va_alist)
va_dcl /* Note: no semicolon */
{
     va_list ap;
     int a1;
     enum arglisttype a2;

     enum argtype ptype;
     int i, *p;
     char c;
     double d;

     /* Initialize the varargs mechanism */
     va_start(ap);

     /* Get the first argument, and arg list flag */
     a1 = va_arg (ap, int);
     a2 = va_arg (ap, enum arglisttype);

     printf ("arg count = %d\n", a1);

     if (a2 == VAR_LIST_PRESENT) {
  /* pick up all the arguments */
  do {
      /* get the type of the argument */
      ptype = va_arg (ap, enum argtype);

      /* retrieve the argument based on the type */
      switch (ptype) {
   case CHAR:  c = va_arg (ap, char);
        printf ("char = %c\n", c);
        break;

   case DOUB:  d = va_arg (ap, double);
        printf ("double = %f\n", d);
        break;

   case PINT:  p = va_arg (ap, int *);
        printf ("pointer = %x\n", p);
        break;

   case INT :  i = va_arg (ap, int);
        printf ("int = %d\n", i);
        break;

   case END_OF_LIST :
        break;
   default:    printf ("bad argument type %d\n", ptype);
        ptype = END_OF_LIST; /* to break loop */
        break;
      } /* switch */
  } while (ptype != END_OF_LIST);
     }

     /* Clean up */
     va_end (ap);
 }

 main()
 {
     int x = 99;

     foo (1, NO_VAR_LIST);
     foo (2, VAR_LIST_PRESENT, DOUB, 3.0, PINT, &x, END_OF_LIST);
 }
```

C9X standard macros

The C9X standard-compliant version of "variadic" or variable argument (varargs) macro notation has been added to the HP ANSI C compiler. The notation for C9X standard and for the GNU version of `varargs` is very similar.

If you have coded your macros to the GNU standards, you can expect GNU-style behavior using the HP ANSI C compiler. If you have coded your macros to C9X standards, you can expect C9X-style behavior.

Usage Differences

In the HP ANSI C compiler, an ellipsis (...) at the end of the parameter list in a macro definition indicates that the macro takes a variable number of arguments, or is variadic in nature. The ellipsis should be the final parameter in C9X style and should immediately follow the final parameter in GNU style.

The last parameter in a variable argument definition is referred to as the variable argument parameter. In GNU terminology, this is known as the *rest* parameter. In a C9X style definition, the variable argument parameter does not explicitly appear in the parameter list but is referred to by the identifier `__VA_ARGS__` in its replacement text.

In the use of a variable arguments macro, a one-to-one correspondence is set up between the arguments in use and those in its definition. This is up to, but not including, the variable argument parameter. The rest of the arguments in the use of the macro definition are referred to as the trailing arguments. For purposes of expanding the macro, the entire combination of the trailing arguments (including the separating commas) is substituted for the variable argument parameter in the replacement text.

There are minor usage differences between how C9X and GNU specify variable argument macros are defined:

Table 10-3 How C9X and GNU define a variable argument macro

C9X
`#define foo(f, ...) printf (f, __VA_ARGS__)`
GNU
`#define foo(f, s...) printf(f, s)`

- In the GNU style, the name of the variable parameter s precedes the ellipsis in the parameter list.
- In the C9X standard, the identifier `__VA_ARGS__` refers to the variable arguments.
- The identifier `__VA_ARGS__` can only occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the arguments.

Variable names

Variable names are also handled slightly different by C9X and GNU.

Table 10-4 How C9X and GNU refer to varargs in macro replacement text

C9X	GNU
Specified by the identifier `__VA_ARGS__`.	Name appears in the replacement text.

For example:

- `__VA_ARGS__` in the replacement text indicates the variable name in the following C9X-style code:

  ```
  printf(f, __VA_ARGS__)
  ```

- s in the replacement text indicates the variable name in the following GNU-style code:

```
printf(f, s)
```

How HP C implements GNU and C9X macros

If you intend to use GNU style variable argument macros in HP C, note that you can make the concatenation operator ## prevent syntax errors from occurring when the variable argument comes in as empty (the null string). However, you can also insert whitespace to the left of the left operand of ## to more accurately specify the intended left operand.

For example, if you use

```
#define foo(f, s...) printf(f, s)
```

Then the macro "call"

```
foo("Hello world.\n");
```

results in the expansion

```
printf("Hello world.\n",);
```

(note the comma ",") causing a syntax error.

GNU provides the following workaround for this kind of a situation. If you use:

```
#define foo(f, s...) printf(f, ## s)
```

If the variable parameter `s` is non-null, if for example, you use:

```
foo("%s %d\n", "Cycles", "1024");
```

the result is

```
printf("%s %d\n", "Cycles", "1024");
```

as the expansion as you would expect.

However, if `s` is null, this erases the comma to the left of the ## in the macro definition and resulting expansion is:

```
printf("Hello world.\n");
```

Note that the comma is gone.

In order to get the same behavior in HP C, you must insert a space to the left of the comma to make it clear to the preprocessor that the comma is the left operand of the '##' operator. Thus your definition for the macro `foo` is:

```
#define foo(f, s...) printf(f , ## s)
```

(Note the space to the left of the ## operator in the macro definition.)

If the space is not inserted, the left operand of the ## operator is understood to be:

```
    printf(f,
```

Because there is no parameter by that name for `foo`, it is erased.

HP Specific Type Qualifiers

See “HP-Specific Type Qualifiers” on page 44.

Location of Files

Table 10-5 lists the location of the HP C files.

Table 10-5 Location of Files

File or Library	Location
Driver	/opt/ansic/bin/cc /opt/ansic/bin/c89
Preprocessor	/opt/langtools/lbin/cpp (Compatibility mode) /opt/langtools/lbin/cpp.ansi (ANSI mode) (Only used with +legacy_cpp)
Compiler	/opt/ansic/lbin/ecom (for A.06.*) /opt/ansic/lbin/ctcom (for A.05.*)
Assembler	/usr/ccs/bin/as
Linker	/usr/ccs/bin/ld
32-bit dynamic loader	/usr/lib/hpux32/dld.so
64-bit dynamic loader	/usr/lib/hpux64/dld.so
Advanced Optimizing Code Generator	/opt/langtools/lbin/u2comp /opt/langtools/lbin/be /opt/langtools/lib/hpux##/libu2comp.so (plugin) (## is 32 or 64 - provided as part of the HP-UX core system)
C libraries (libc)	/usr/lib/hpux##/libc.so (## is 32 or 64)
lex and yacc libraries	/usr/lib/hpux##/libl.so /usr/lib/hpux##/liby.so (## is 32 or 64)

Table 10-5 Location of Files (Continued)

File or Library	Location

Manpages	• Manpages in English
	`/opt/ansic/share/man/man1.Z/cc.1`
	`/opt/ansic/share/man/man1.Z/c89.1`
	`/opt/ansic/share/man/man1.Z/c99.1`
	`/opt/langtools/share/man/man1.Z/cpp.1`
	`/opt/langtools/share/man/man1.Z/lex.1`
	`/opt/langtools/share/man/man1.Z/yacc.1`
	• Manpages in Japanese
	`/opt/ansic/share/man/ja_JP.SJIS/man1.Z/cc.1`
	`/opt/ansic/share/man/ja_JP.SJIS/man1.Z/c89.1`
	`/opt/ansic/share/man/ja_JP.SJIS/man1.Z/c99.1`
	`/opt/ansic/share/man/ja_JP.eucJP/man1.Z/cc.1`
	`/opt/ansic/share/man/ja_JP.eucJP/man1.Z/c89.1`
	`/opt/ansic/share/man/ja_JP.eucJP/man1.Z/c99.1`

Message Catalogs	`/opt/langtools/lib/nls/msg/C/cpp.cat`
	`/opt/langtools/lib/nls/msg/C/lex.cat`
	`/opt/langtools/lib/nls/msg/C/yacc.cat`
	`/opt/ansic/lib/nls/msg/C/aCC.cat`
	`/opt/ansic/lib/nls/msg/C/ecc.cat (A.06.*)`

`prof` and `gprof` libraries	`/usr/lib/hpux##/libprof.so`
	`/usr/lib/hpux##/libgprof.so`
	(## is 32 or 64)

Temporary files	`/var/tmp`*

Math libraries	`/usr/lib/hpux##/libm.a`
	`/usr/lib/hpux##/libm.so`
	(## is 32 or 64)

OpenMP libraries	`/usr/lib/hpux##/libomp.a`
	`/usr/lib/hpux##/libomp.so`
	`/usr/lib/hpux##/libcps.a`
	`/usr/lib/hpux##/libcps.so`
	(## is 32 or 64)

Release Notes	`/opt/ansic/newconfig/RelNotes/ACXX.release.notes`

Table 10-5 Location of Files (Continued)

File or Library	Location
Online Help	`/opt/ansic/html/C/guide`
C Tools	`/opt/langtools/bin` `/opt/langtools/lbin`

*You can change the default location for the temporary files used and created by the C compiler by setting the environment variable `TMPDIR`. If the compiler cannot write to `$TMPDIR`, it uses the default location `/var/tmp`. See the *HP-UX Reference* for details.

A Syntax Summary

This appendix presents a summary of the C language syntax using a variation of the Backus Naur syntax.

Lexical Grammar

Tokens

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

```
preprocessing-token ::=
        header-name
        identifier
        pp-number
        character-constant
        string-literal
        operator
        punctuator
        each non-white-space character cannot be one
of the above
```

Keywords

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

Identifiers

```
identifier ::= nondigit
                 identifier nondigit
                 identifier digit
                 identifier dollar-sign

nondigit ::= any character from the set:
           _ a b c d e f g h i j k l m n o p
           q r s t u v w x y z A B C D E F G
           H I J K L M N O P Q R S T U V W X
           Y Z

digit ::= any character from the set:
        0 1 2 3 4 5 6 7 8 9

dollar-sign ::= the $ character
```

Constants

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

floating-constant ::=
       fractional-constant [exponent-part] [floating-suffix]
       digit-sequence exponent-part [floating-suffix]

fractional-constant ::=
       [digit-sequence] . digit-sequence
       digit-sequence .

exponent-part ::=
    e [sign] digit-sequence
    E [sign] digit-sequence

sign ::=
    +
    -
```

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

floating-suffix ::=
    f  l  F  L

integer-constant ::=
    decimal-constant [integer-suffix]
    octal-constant [integer-suffix]
    hexadecimal-constant [integer-suffix]

decimal-constant ::=
    nonzero-digit
    decimal-constant digit

octal-constant ::=
    0
    octal-constant octal-digit

hexadecimal-constant ::=
    0x hexadecimal-digit
    0X hexadecimal-digit
    hexadecimal-constant hexadecimal-digit

nonzero-digit ::= any character from the set:
    1  2  3  4  5  6  7  8  9

octal-digit ::= any character from the set
    0  1  2  3  4  5  6  7

hexadecimal-digit ::= any character from the set
    0  1  2  3  4  5  6  7  8  9
    a  b  c  d  e  f
    A  B  C  D  E  F


integer-suffix ::=
    unsigned-suffix [long-suffix]
    length-suffix [unsigned-suffix]
```

```
unsigned-suffix ::=
    u  U

length-suffix ::=
    long-suffix
    long-long-suffix

long-suffix ::= any character from the set
    l  L

long-long-suffix ::= any character from the set
    ll LL Ll lL

enumeration-constant ::= identifier

character-constant ::=
    'c-char-sequence'
    L'c-char-sequence'

c-char-sequence ::=
    c-char
    c-char-sequence c-char

c-char ::=
    any character in the source character set except
       the single quote ('), backslash (\), or new-line character
    escape-sequence

escape-sequence ::=
    simple-escape-sequence
    octal-escape-sequence
    hexadecimal-escape-sequence

simple-escape-sequence ::=
    \'   \"    \?  \\   \ddd   \xdd
    \a   \b    \f   \n   \r   \t   \v

octal-escape-sequence ::=
    \ octal-digit
    \ octal-digit octal-digit
    \ octal-digit octal-digit octal-digit
```

```
hexadecimal-escape-sequence ::=
     \x hexadecimal-digit
     hexadecimal-escape-sequence hexadecimal-digit
```

String Literals

```
string-literal ::=
    "[s-char-sequence]"
    L"[s-char-sequence]"

s-char-sequence ::=
    s-char
    s-char-sequence s-char

s-char ::=
    any character in the source character set except
           the double-quote (") , backslash (\), or new-line
    character escape-sequence
```

Operators

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

Punctuators

```
punctuator ::=  One selected from:
    [   ]   (   )   {   }   *   ,   :   =   ;   ...   #
```

Header Names

```
header-name ::=
         <h-char-sequence>
         "q-char-sequence"

h-char-sequence ::=
         h-char
         h-char-sequence h-char

h-char ::=
         any character in the source character set except
```

```
            the newline character and >

q-char-sequence ::=
         q-char
         q-char-sequence q-char

q-char ::=
         any character in the source character set except
             the newline character and "
```

Preprocessing Numbers

```
pp-number ::=
         digit
         . digit
         pp-number digit
         pp-number nondigit
         pp-number e sign
         pp-number E sign
         pp-number .
```

Phrase Structure Grammar

Expressions

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

postfix-expression ::=
        primary-expression
        postfix-expression  [ expression ]
        postfix-expression ( [argument-expression-list] )
        postfix-expression . identifier
        postfix-expression -> identifier
        postfix-expression ++
        postfix-expression --

argument-expression-list ::=
        assignment-expression
        argument-expression-list , assignment-expression

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


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

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

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

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

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

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

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


AND-expression  ::=
        equality-expression
        AND-expression & equality-expression

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

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

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

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

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

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

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


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

constant-expression  ::=
        conditional-expression
```

Declarations

```
declaration ::=
     declaration-specifiers [init-declarator-list] ;

declaration-specifiers ::=
     storage-class [declaration-specifiers]
     type-specifier [declaration-specifiers]
     type-qualifier [declaration-specifiers]

init-declarator-list ::=
     init-declarator
     init-declarator-list , init-declarator

init-declarator ::=
     declarator
     declarator = initializer

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

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

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

struct-or-union ::=
      struct
      union

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

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

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

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

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

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

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

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

type-qualifier ::=
      const
      noalias
      volatile

declarator  ::=
      [pointer] direct-declarator

direct-declarator ::=
      identifier
      ( declarator )
      direct-declarator [ [constant-expression] ]
      direct-declarator ( parameter-type-list )
      direct-declarator ( [identifier-list] )

pointer ::=
     * [type-qualifier-list]
     * [type-qualifier-list] pointer

type-qualifier-list ::=
     type-qualifier
     type-qualifier-list type-qualifier
```

```
parameter-type-list ::=
     parameter-list
     parameter-list , ...

parameter-list ::=
     parameter-declaration
     parameter-list , parameter-declaration

parameter-declaration ::=
     declaration-specifiers declarator
     declaration-specifiers [abstract-declarator]

identifier-list ::=
     identifier
     identifier-list , identifier

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

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

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

typedef-name ::=
     identifier

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

initializer-list ::=
     initializer
     initializer-list , initializer
```

Statements

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

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

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

declaration-list :=
     declaration
     declaration-list declaration

statement-list :=
     statement
     statement-list statement

expression-statement :=
     [expression];

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

iteration-statement :=
     while ( expression ) statement
     do  statement while ( expression )
     for ([expression]; [expression]; [expression] ) statement

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

External Definitions

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

external-declaration :=
     function-definition
     declaration

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

Preprocessing Directives

```
preprocessing-file :=
     [group]

group :=
     group-part
     group group-part

group-part :=
     [pp-tokens] new-line
     if-section
     control-line

if-section :=
     if-group [elif-groups] [else-group] endif-line

if-group :=
     # if     constant-expression new-line [group]
     # ifdef  identifier new-line [group]
     # ifndef identifier new-line [group]

elif-groups :=
     elif-group
     elif-groups elif-group

elif-group :=
     # elif   constant-expression new-line [group]

else-group :=
     # else   new-line [group]

endif-group :=
     # endif  new-line

control-line :=
     # include pp-tokens new-line
     # define  identifier replacement-list new-line
     # define  identifier([identifier-list] ) replacement-list  newline
     # undef   identifier new-line
     # line    pp-tokens new-line
     # error   [pp-tokens] new-line
     # pragma  [pp-tokens] new-line
     #         new-line

replacement-list :=
     [pp-tokens]

pp-tokens :=
     preprocessing-token
     pp-tokens preprocessing-token

new-line :=
     the new-line character
```

Index

Index