# HP C/iX Reference Manual

## HP 3000 MPE/iX Computer Systems

**Edition 3**

HEWLETT® PACKARD

**Manufacturing Part Number: 31506-90011**
**E0499**

U.S.A. April 1999

# 1 Introduction

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

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

In December of 1989, the ANSI board approved the *American National Standard for Programming Language C, X3.159*.

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

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

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

Many programs compile and execute properly in an ANSI C environment with no changes. In the vast majority of cases where a change is required, the offending construct will be identified by a warning or error message produced by the compiler. In a few cases, which are believed to be rare in actual practice, certain program constructs will be accepted but will behave differently under ANSI C. HP C/iX is capable of producing migration warnings to help identify code where such "quiet changes" would occur.

## ANSI Mode

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

Because an ANSI-conforming compiler is required to do more thorough error detection and reporting than has been traditional among C compilers, 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 the major new feature, function prototypes.

## Non-ANSI 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 non-ANSI mode of compilation has been provided. In this mode, virtually all programs that compiled and executed under previous releases of HP C/iX will continue to work as expected.

If you do not specify the mode of compilation, non-ANSI mode is the default. However, the default mode will be changed to ANSI on some future release.

## Focus of This Manual

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

HP C/iX, when invoked in ANSI mode, is intended to be a conforming implementation of ANSI C, as specified by American National Standard X3.159. 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/iX. 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.

# 2 Lexical Elements

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

## Tokens

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

## Syntax

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

## Description

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

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

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

For example:

```
a-  -1
```

is different from

```
a1
```

## Keywords

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

```
auto        do          goto        signed    union
break       double      if          sizeof    unsigned
case        else        int         static    void
char        enum        long        sizeof    volatile
const       extern      register    struct    while
continue    float       return      switch
default     for         short       typedef
```

## Identifiers

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

### Syntax

```
identifier := nondigit
            identifier nondigit
            identifier digit

nondigit := any character from the set:
            _ a b c d e f g h i j k l m n o p
            q r s t u v w x y z A B C D E F G
            H I J K L M N O P Q R S T U V W X
            Y Z

digit := any character from the set:
         0 1 2 3 4 5 6 7 8 9

dollar-sign := the $ character
```

#### Description

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

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

An HP extension to the language non-ANSI mode allows $ as a valid character in an identifier as long as it is not the first character. The following are examples of legal and illegal identifiers:

#### Legal

```
Sub_Total
X
aBc
Else
do_123
```

#### Illegal

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

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

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

## Identifier Scope

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

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

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

## Identifier Linkage

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

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

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

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

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

If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier `extern`. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

The following identifiers have no linkage:

- An identifier declared to be anything other than an object or a function.
- An identifier declared to be a function parameter.
- A block scope identifier for an object declared without the storage-class specifier `extern`.

For example:

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

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

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

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

```
{
   int i;       <-A
     .
     .          <-B
     .
   {
      float i;  <-C
         .      <-D
         .
         .
   }            <-E
   .
   .            <-F
   .
}               <-G
```

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

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

## Storage Duration

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

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

## Name Spaces

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

Four different name spaces are used in C:

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

functions, `typedef` names, and enumeration constants.

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

## Types

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

1. **Object Type**

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

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

      1. **Pointer** — These types include pointers to objects and functions.
      2. **Arithmetic** — These types include floating and integral types.

         - **Floating**: The floating types include the following:

           **Float** — A 32-bit floating point number. **Double** — A 64-bit double precision floating point number. **Long double** — A 128-bit quad precision floating point number.

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

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

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

           **Enumerated** — Enumerated types are explicitly listed by the programmer; they name specified integer constant values. The enumerated type `color` might, for example, define `red`,
           `blue`, and `green`. An object of type `enum color` could then have the value `red`, `blue`, or `green`.

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

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

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

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

2. **Function Type**

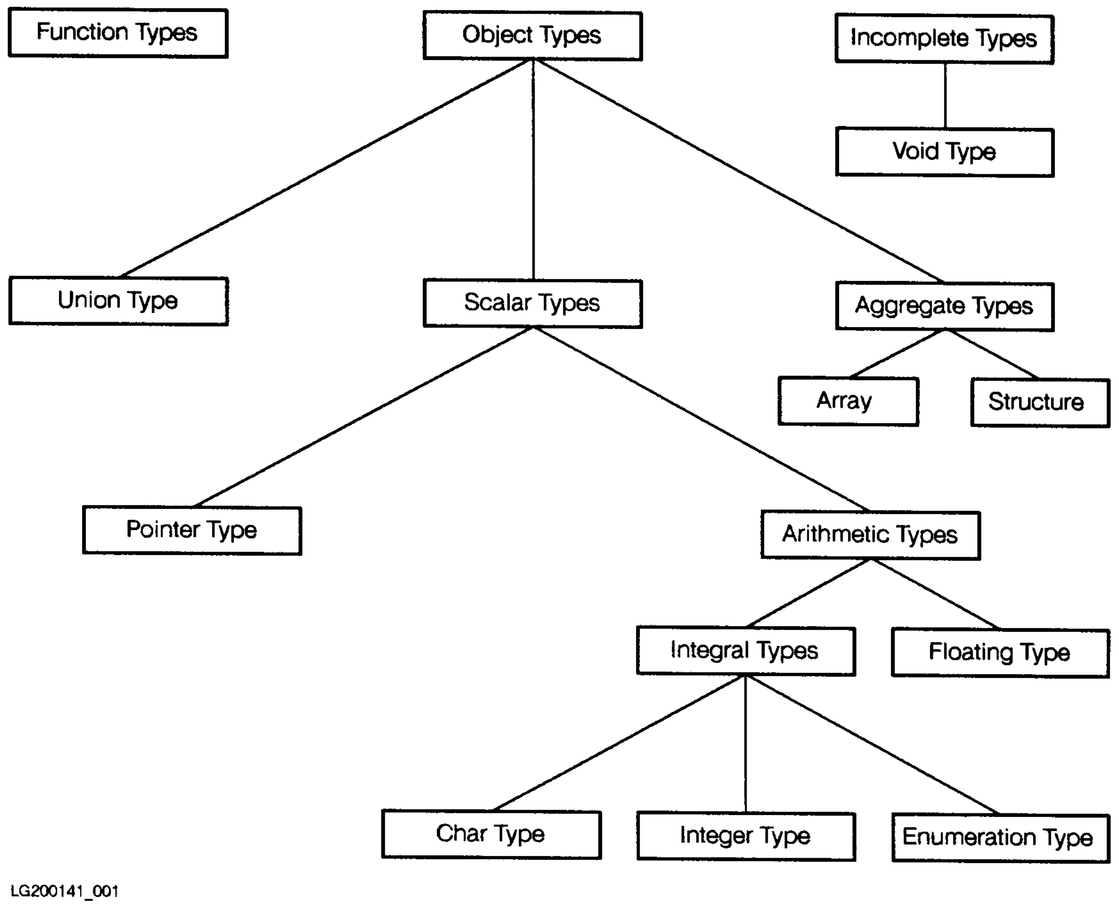
   A function type specifies the type of the object that a function returns. A function that returns an object of type T can be referred to as a "function returning T," or simply, a T function.

3. **Incomplete Type**

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

Figure 2-1 illustrates the C types.

**Figure 2-1. C Types**



## Constants

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

### Syntax

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

### Description

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

```
2+8/2
```

are automatically interpreted as a single constant at compile time.

### Floating Constants

Floating constants represent floating-point values.

#### Syntax

```
floating-constant   :=
     fractional-constant [exponent-part] [floating-suffix]
     digit-sequence exponent-part [floating-suffix]
fractional-constant   :=
     [digit-sequence] . digit-sequence
     digit-sequence .

exponent-part  :=
     e [sign] digit-sequence
     E [sign] digit-sequence

sign :=
     +
     -

digit-sequence  :=
     digit
     digit-sequence digit

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

**NOTE** Suffixes in floating-constants are available only in ANSI mode.

### Description

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

The format of floating-point numbers is given in Chapter 9, HP C/iX Implementation Topics.

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

### Examples

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

## Integer Constants

*Integer constants* represent integer values.

### Syntax

```
integer-constant :=
        decimal-constant [integer-suffix]
        octal-constant [integer-suffix]
        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-sufix]
        long-suffix [unsigned-suffix]

unsigned-suffix := any character from the set
        u  U

long-suffix := any character from the set
        l  L
```

**NOTE** The u and U suffixes are available only in ANSI mode.

## Description

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

The size and type of integer constants are described in Chapter 9, HP C/iX Implementation Topics.

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

```
077   01L   01234567 0222l
```

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

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

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

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

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

## Examples

```
0xFFFFu     unsigned int, hexadecimal
4196L       signed long int, decimal
0X89ab      signed int, hexadecimal
047L        signed long int, octal
64U         unsigned int, decimal
15          signed int, decimal
15L         signed long int, decimal
15U         unsigned int, decimal
15UL        unsigned long int, decimal
15LU        unsigned long int, decimal
0125        signed int, octal
```

## Enumeration Constants

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

### Syntax

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

### Description

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

## Character Constants

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

### Syntax

```
character-constant:
         'c-char-sequence'
        L'c-char-sequence'

c-char-sequence:
        c-char
        c-char-sequence c-char

c-char:
        any character in the source character set except
            the single-quote ', backslash \, or new-line character
        escape-sequence

escape-sequence:
        simple-escape-sequence
        octal-escape-sequence
        hexadecimal-escape-sequence

simple-escape-sequence: one of
        \'  \"  \?  \\
        \a  \b  \f  \n  \r  \t  \v

octal-escape-sequence:
        \ octal-digit
        \ octal-digit octal-digit
        \ octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:
        \x hexadecimal-digit
        hexadecimal-escape-sequence hexadecimal-digit
```

**NOTE** \a and \? are available only in ANSI mode.

### Description

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

Integral character constants are of type `int`. They do not have type `char`. However, because a `char` is normally converted to an `int` in an expression, this seldom is a problem. The contents can be ASCII characters, octal escape sequences, or hexadecimal escape sequences. Octal escape sequences consist of a backslash, ( \ ) followed by up to three octal digits. Hexadecimal escape sequences also start with a backslash, which is followed by lowercase x and any number of hexadecimal digits. It is terminated by any non-hexadecimal characters. The digits of the escape sequences are converted into a single 8-bit character and stored in the character constant at that point. For example, the following character constants have the same value:

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

They all represent the decimal value 65.

Character constants are not restricted to one character; multi-character character constants are allowed. The value of an integral character constant containing more than one character is computed by concatenating the 8-bit ASCII code values of the characters, with the leftmost character being the most significant. For example, the character constant `'AB'` has the value `256*'A'+'B' = 256*65+66 = 16706`. Only the rightmost four characters participate in the computation.

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

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

Some characters are given special representation in escape sequences. These are nonprinting and special characters that programmers often need to use (listed in <Undefined Cross-Reference> below).

**Table 2-1. Special Characters**

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

## Examples

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

## String Literals

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

### Syntax

```
string-literal:
        "[s-char-sequence]"
        L"[s-char-sequence]"

s-char-sequence:
      s-char
      s-char-sequence s-char

s-char:
        any character in the source character set except
               double quote, backslash, or newline
        escape-sequence
```

### Description

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

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

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

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

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

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

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

### Examples

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

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

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

## Operators

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

### Syntax

```
operator := One selected from the set
```

### Description

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

is parsed as if it had been written

which results in a syntax error. An alternate parse

is not chosen because it does not follow the longest first rule, even though it results in a syntactically correct expression. As a result, white space is often important in writing expressions that use complex operators. The precedence of operators is discussed in more detail in Chapter 5.

The obsolete form of the assignment operators (=* instead of *=) is not supported. If this form is used, the compiler parses it as two tokens (= and *).

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

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

## Punctuators

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

### Syntax

```
         punctuator :=  One selected from:
               [   ]   (   )   {   }   *   ,   :   =   ;   #   ...
```

### Description

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

### Example

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

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

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

## Comments

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

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

```
#if 0
    .
    .
   code
    .
    .
#endif
```

# 3 Data Types and Declarations

In C, as in many other programming languages, you must declare identifiers before you can use them. The declarable entities in C are objects; functions; tags and members of structures, unions, and enumerated types; and typedef names. This chapter describes declarations, type specifiers, storage-class specifiers, structure and union specifiers, enumerations, declarators, type names, and initialization. Data types and declarations are defined using Backus-Naur form.

## Program Structure

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

### Syntax

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

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

### Description

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

## Declarations

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

### Syntax

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

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

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

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

### Description

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

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

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

Function definitions have a slightly different syntax as discussed in 'Function Declarators' later in this chapter. Also, note that it is often valid to define a tag (`struct`, `union`, or `enum`) without actually declaring any objects.

### Examples

#### Valid Declarations:

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

```
const float pi = 3.14;                 /* a constant float, initialized   *
/
const float *const pi_ptr = &pi;       /* a constant pointer to a constant
                                          float, initialized with an
                                          address constant                *
/
static j1, j2, j3;                     /* initialized to zero by default  *
/
typedef struct
   {double real, imaginary;} Complex;   /* declares a type name            *
/
Complex impedance = {47000};           /* second member defaults to zero  *
/
enum color {red=1, green, blue};       /* declares an enumeration tag and
                                          three constants                 *
/
int const short static volatile signed
   really_Strange = {sizeof '\?'};      /* pretty mixed up                 *
/
```

**Invalid Declarations:**

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

## Storage-Class Specifiers

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

### Syntax

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

### Description

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

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

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

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

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

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

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

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

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

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

## Type Specifiers

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

### Syntax

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

### Description

Most of the type specifiers are single keywords. (Refer to chapter 9 for sizes of types.) The syntax of the type specifiers permits more types than are actually allowed in the C language. The various combinations of type specifiers that are allowed are shown in Table 3-1. Type specifiers that are equivalent appear together in a box. For example, specifying `unsigned` is equivalent to `unsigned int`. Type specifiers may appear in any order, possibly intermixed with other declaration specifiers.

**Table 3-1. C Type Specifiers**

| |
|---|
| void |
| char |
| signed char |
| unsigned char |
| short, signed short, short int, or signed short int |
| unsigned short, or unsigned short int |
| int, signed, signed int, or *no type specifiers* |
| unsigned, or unsigned int |
| long, signed long, long int, or signed long int |
| unsigned long, or unsigned long int |
| float |

**Table 3-1. C Type Specifiers**

| |
|---|
| double |
| long double |
| *struct-or-union specifier* |
| *enum-specifier* |
| *typedef-name* |

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

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

## Type Qualifiers

### Syntax

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

### Description

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

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

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

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

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

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

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

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

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

| Declaration | Meaning |
|---|---|
| volatile int vol_int; | Delclares a volatile int variable. |
| const int *ptr_to_const_int;<br>int const *ptr_to_const_int; | Both declare a variable pointer to a constant int. |
| int *const const_ptr_to_int; | Declares a constant pointer to a variable int. |
| int *volatile vpi, *pi; | Declares two pointers: vpi is a volatile pointer to an int; pi is a pointer to an int. |

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

| Declaration | Meaning |
|---|---|
| int const *volatile vpci; | Declares a volatile pointer to a constant int. |
| const *pci; | Declares a pointer to a constant int. Since no type specifier was given, it defaults to int. |

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

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

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

```
volatile int *ptr_to_vol_int;
```

the type would be `pointer to volatile int`.

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

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

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

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

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

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

## Structure and Union Specifiers

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

### Syntax

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

struct-or-union ::=
     struct
     union

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

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

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

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

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

### Description

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

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

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

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

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

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

A colon followed by an integer constant expression indicates that the compiler should create an unnamed bit-field at that location. In addition, a colon followed by a zero indicates that the current location is full and that the next bit-field should begin at the start of the next storage location. Refer to chapter 9 for the treatment of the sign for bit-fields. Although bit-fields are permitted in unions (ANSI mode only), they are just like any other members of the union in that they have a zero offset from the beginning of the union. That is, they are not packed into the same word, as in the case of structures. The special cases of unnamed bit-fields and unnamed bit-fields of length zero behave differently with unions; they are simply unnamed members that cannot be assigned to.

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

The largest bit-field is the length of an `int` (refer to chapter 9 for exact sizes). Bit-fields do not straddle a word boundary.

## Structure and Union Tags

Structures and unions are declared with the struct or union keyword. You can follow the keywords with a tag that names the structure or union type much the same as an enum tag names the enumerated type. (Refer to the section 'Enumeration' later in this chapter for information on enumerated types.) Then you can use the tag with the struct or union keyword to declare variables of that type without respecifying 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, which point to an object 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 ::=
   enum[ identifier] {enumerator-list}
   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. Note that `enum` constant names must be unique.

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

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

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

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

## Examples

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

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

## 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 ints. This is because the array declarator binds more tightly than the pointer declarator. The declaration

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

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

## Pointer Declarators

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

Some examples:

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

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

## Array Declarators

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

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

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

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

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

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

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

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

Following are examples of array declarations:

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

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

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

## Function Declarators

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

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

or

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

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

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

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

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

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

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

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

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

## Type Names

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

### Syntax

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

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

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

### Description

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

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

### Examples

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

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

## Type Definitions Using typedef

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

### Syntax

*typedef-name* ::= *identifier*

### Description

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

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

```
  typedef int *pointer;
```

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

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

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

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

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

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

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

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

Other type specifiers (`void, char, short, int, long, signed, unsigned, float,` and `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
```

### Examples

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

You initialize structures as you do any other aggregate:

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

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

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

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

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

### 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 runtime 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 elided. For example,

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

is normally specified as

```
int i = 3;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

## Example

```
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.

Union initialization is only available in ANSI mode.

### Example

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

## Function Definitions

A *function definition* introduces a new function.

### Syntax

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

### Description

A function definition provides the following information about the function:

1. **Type**. You can specify the return type of the function. If no type is provided, the default return type is `int.` If the function does not return a value, it can be defined as having a return type of `void`. You can declare functions as returning any type except a function or an array. You can, however, define functions that return pointers to functions or pointers to arrays.
2. **Formal parameters**. There are two ways of specifying the type and number of the formal parameters to the function:
   a. A function declarator containing an *identifier list*

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

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

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

**NOTE** Function prototypes can be used only in ANSI mode.

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

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

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

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

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

**NOTE** In non-ANSI 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 converted to pointer to type T, and functions are converted 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.   */
    }
```

# 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 can be used in an expression, a narrower integral type can 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, 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, non-ANSI 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 non-ANSI mode. The following program shows two expressions that are evaluated differently in the two modes.

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

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.

In non-ANSI mode, as with many pre-ANSI compilers, the results will be:

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

ANSI C gives the following results:

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

## Usual Arithmetic Conversions

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

1. If either operand is long double, the other operand is converted to long double.
2. If either operand is double, the other operand is converted to double.
3. If either operand is float, the other operand is converted to float.
4. Integral promotions are performed on both operands, and then the rules listed below are followed.
   a. If either operand is unsigned long int, the other operand is converted to unsigned long int.
   b. If one operand is long int and the other is unsigned int, both operands are converted to unsigned long int.
   c. If either operand is long int, the other operand is converted to long int.
   d. If either operand is unsigned int, the other operand is converted to unsigned int.
   e. Otherwise, both operands have type int.

**NOTE** In non-ANSI mode, the rules are slightly different.

Rule 1: Does not apply, because long double is not supported in non-ANSI mode.

Rule 3: Does not apply, because in non-ANSI mode, whenever a float appears in an expression, it is immediately converted to a double.

Rule 4: The integral promotions are performed according to the unsigned preserving rules when compiling in non-ANSI 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.

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 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 &.
- When the array is a character string literal initializing an array of characters.
- When the array is a wide string literal initializing an array of wide characters.

# 5 Expressions

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

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

The C language does not define the evaluation order of subexpressions within a larger expression except in the special cases of the &&, `||`, `?:`, and `,` operators. When programming in other computer languages, this may not be a concern. C's rich operator set, however, introduces operations that produce "side effects." The ++ operator is a prime example. The ++ operator increments a value by 1 and provides the value for further calculations. For this reason, expressions such as

```
b = ++a*2 + ++a*4;
```

are dangerous. The language does not specify whether the variable `a` is first incremented and multiplied by 4 or is first incremented and multiplied by 2. The value of this expression is undefined.

## Operator Precedence

*Precedence* is the order in which the compiler groups operands with operators. The C compiler evaluates certain operators and their operands before others. If operands are not grouped using parentheses, the compiler groups them according to its own rules.

<Undefined Cross-Reference> shows the rules of operator precedence in the C language. <Undefined Cross-Reference> lists the highest precedence operators first. Most operators group from the left to the right but some group from the right to the left. The grouping indicates how an expression containing several operators of the same precedence will be evaluated. Left to right grouping means the expression

```
a/b * c/d
```

behaves as if it had been written:

```
a/b)*c)/d)
```

Likewise, an operator that groups from the right to the left causes the expression

```
a = b = c
```

to behave as if it had been written:

```
a = (b = c)
```

**Table 5-1. C Operator Precedence**

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

**Table 5-1. C Operator Precedence**

| Operators | Grouping |
|---|---|
| `= *= /= %= += -= <<= >>= &= ^= \|=` | right to left |
| `,` | left to right |

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

## Lvalue Expressions

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

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

### Examples

Given the following declarations:

```
int *p, a, b;
int arr[4];
int func();

   a                     /* Lvalue */
   a + b                 /* Not an lvalue */
   p                     /* Lvalue */
   *p                    /* Lvalue */
   arr                   /* Lvalue, but not modifiable */
   *(arr + a)            /* Lvalue */
   arr[a]                /* Lvalue, equivalent to *(arr+a) */
   func                  /* Not an lvalue */
   func()                /* Not an lvalue */
```

## Primary Expressions

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

### Syntax

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

### Description

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

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

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

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

### Examples

```
identifier:  var1

constant: 99

string-literal:  "hi there"

( expression )  (ab)
```

## Postfix Operators

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

### Syntax

```
postfix-expression :=
       primary-expression
       postfix-expression [ expression ]
       postfix-expression ( [argument-expression-list] )
       postfix-expression . identifier
       postfix-expression -> identifier
       postfix-expression
       postfix-expression

argument-expression-list :=
       assignment-expression
       argument-expression-list , assignment-expression
```

### Examples

The following are examples of postfix operators:

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

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

The postfix decrement operator () : index

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

The selection operator (.)          : struct_name.member

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

## Array Subscripting

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

### Syntax

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

### Description

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

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

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

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

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

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

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

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

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

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

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

## Function Calls

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

### Syntax

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

### Description

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

Although the expression denoting the called function must actually be a *pointer* to a function, in typical usage, it is simply a function name. This works because the function name will automatically be converted to a pointer, as explained in chapter 4.

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

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

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

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

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

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

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

In this case, the compiler does not do any checking between the argument types and the types of the parameters of the function (even if it has seen the definition of the function).

Thus, for safety, it is highly advisable to use prototypes wherever possible.

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

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

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

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

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

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

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

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

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

## Structure and Union Members

A member of a structure or a union can be referenced using either of two operators: the period or the right arrow.

### Syntax

```
postfix-expression . identifier
postfix-expression -> identifier
```

### Description

Use the period to reference members of structures and unions directly. Use the arrow operator to reference members of structures and unions pointed to by pointers. The arrow operator combines the functions of indirection through a pointer and member selection. If `P` is a pointer to a structure with a member `M`, the expression `P->M` is identical to `(*P).M.`

The *postfix-expression* in the first alternative must be a structure or a union. The expression is followed by a period (**.**) and an identifier. The identifier must name a member defined as part of the structure or union referenced in the *postfix-expression*. The value of the expression is the value of the named member. It is an lvalue if the *postfix-expression* is an lvalue.

If the *postfix-expression* is a pointer to a structure or a pointer to a union, follow it with an arrow (composed of the `-` character followed by the `>`) and an identifier. The identifier must name a member of the structure or union which the pointer references. The value of the primary expression is the value of the named member. The resulting expression is an lvalue.

The **.** operator and the **->** operator are closely related. If `S` is a structure, `M` is a member of structure `S`, and  is a valid pointer expression, `S.M` is the same as `(S)->M`.

## Postfix Increment and Decrement Operators

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

### Syntax

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

### Description

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

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

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

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

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

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

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

## Unary Operators

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

### Syntax

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

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

### Examples

*The unary plus operator*: `+var`

*The unary minus operator*:  `-var`

*The address-of operator*: `&var`

*The indirect operator*: `*ptr`

*The logical NOT operator*: `!var`

*The bitwise NOT operator*: `~var`

## Prefix Increment and Decrement Operators

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

### Syntax

```
 unary-expression
 unary-expression
```

### Description

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

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

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

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

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

## Address and Indirection Operators

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

### Syntax

```
 cast-expression
* cast-expression
```

### Description

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

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

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

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

## Unary Arithmetic Operators

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

### Syntax

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

### Description

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

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

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

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

## The sizeof Operator

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

### Syntax

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

### Description

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

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

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

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

## Cast Operators

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

### Syntax

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

### Description

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

The result of a cast operation is not an lvalue.

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

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

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

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

## Multiplicative Operators

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

### Syntax

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

### Description

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

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

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

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

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

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

**Table 5-2. C Operator Precedence**

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

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

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

**Table 5-3. C Operator Precedence**

| | **b positive** | **b negative** |
|---|---|---|
| **a positive** | + | + |
| **a negative** | - | - |

For example:

```
     -5  2  -1
```

## Examples

```
var1 * var2

var1 / var2

var1  var2
```

## Additive Operators

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

### Syntax

```
additive-expression :=
    multiplicative-expression
    additive-expression  multiplicative-expression
    additive-expression - multiplicative-expression
```

### Description

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

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

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

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

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

### Examples

```
var1+var2
```

```
var1-var2
```

## Bitwise Shift Operators

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

### Syntax

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

### Description

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

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

### Example

```
var1>>var2
```

## Relational Operators

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

### Syntax

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

### Description

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

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

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

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

### Examples

```
var1 < var2

var1 > var2

var1 <= var2

var1 >= var2
```

## Equality Operators

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

### Syntax

```
equality-expression :=
   relational-expression
   equality-expression  relational-expression
   equality-expression != relational-expression
```

### Description

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

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

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

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

```
  if(ab) return 0;

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

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

### Examples

```
var1==var2

var1!=var2
```

## Bitwise AND Operator

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

### Syntax

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

### Description

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

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

**Table 5-4. C Operator Precedence**

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

### Examples

```
var1 & var2
```

## Bitwise Exclusive OR Operator

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

### Syntax

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

### Description

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

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

**Table 5-5. C Operator Precedence**

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

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

### Example

```
var1 ^ var2
```

## Bitwise Inclusive OR Operator

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

### Syntax

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

### Description

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

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

**Table 5-6. C Operator Precedence**

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

### Example

```
var1 | var2
```

## Logical AND Operator

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

### Syntax

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

### Description

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

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

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

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

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

### Example

```
var1 && var2
```

## Logical OR Operator

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

### Syntax

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

### Description

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

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

### Example

```
var1 || var2
```

## Conditional Operator

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

### Syntax

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

### Description

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

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

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

1. **Both arithmetic.**
   - The usual arithmetic conversions are performed on the second and third expressions. The resulting type of the conditional expression is the result of the conversion.
2. **Both are pointers to type T.**
   - Arrays are converted to pointers, if necessary. The result is a pointer to type T.
3. **Identical type object.**
   - The types can match and be structure, union, or `void`. The result is that specific type.
4. **Pointer and Null pointer constant or a pointer to void**
   - One expression may be a pointer (or array that is converted to a pointer) and the other a null pointer constant or a pointer to void. The result is the same type as the type of the pointer operand.

In all cases, the result is not an lvalue.

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

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

## Example

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

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

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

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

## Assignment Operators

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

### Syntax

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

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

### Description

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

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

### Simple Assignment

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

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

1. **Both arithmetic**

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

2. **Both structure/union**

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

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

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

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

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

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

## Compound Assignment

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

```
A op= B
```

the result is equal to the following assignment

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

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

Therefore,

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

is very different from

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

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

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

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

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

## Examples

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

## Comma Operator

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

### Syntax

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

### Description

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

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

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

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

### Examples

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

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

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

## Constant Expressions

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

### Syntax

```
constant-expression ::=
     conditional-expression
```

### Description

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

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

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

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

### Examples

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

# 6 Statements

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

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

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

### Syntax

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

### Example

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

## Labeled Statements

Labeled statements are those preceded by a label.

### Syntax

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

### Description

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

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

### Example

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

## Compound Statement or Block

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

### Syntax

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

declaration-list ::=
     declaration
     declaration-list declaration

statement-list ::=
     statement
     statement-list statement
```

### Description

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

Except when declared as `extern`, variables and constants declared in a block are local to that block and any subordinate blocks declared therein. If the objects are initialized, the initialization is performed each time the compound statement is entered from the top through the left brace ({) character. If the statement is entered via a `goto` statement or in a switch statement, the initialization is not performed.

Any object declared with static storage duration is created and initialized when the program is loaded for execution. This is true even if the object is declared in a subordinate block.

### Example

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

   temp = x;
   x = y;
   y = temp;
}
```

## Selection Statements

A selection statement alters a program's execution flow by selecting one path from a collection based on a specified controlling expression. The `if` statement and the `switch` statement are selection statements.

### Syntax

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

### Example

`if (`*expression*`)` *statement:*

```
    if (x<y) x=y;
```

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

```
    if (x<y) x=y; else y=x;
```

`switch (`*expression*`)` *statement:*

```
    switch (x)
    { case 1: x=y;
              break;
      default: y=x;
              break;
    }
```

## The if Statement

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

### Syntax

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

### Description

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

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

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

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

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

The "dangling else" problem associated with `if` statements of this form is resolved by associating the `else` with the last lexically preceding `if` (without an `else`) that is in the same block, but not in an enclosed block.

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

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

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

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

## The switch Statement

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

### Syntax

```
 switch (expression)
     statement
```

### Description

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

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

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

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

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

The `switch` statement operates like a multiway `else-if` chain except the values in the `case` statements must be constant expressions, and, most importantly, once a statement is selected from within the `switch body`, control is passed from statement to statement as in a normal C program. Control may "fall" through to following case statements. Using a `break` statement is the most common way to leave a switch body. If a `break` statement is encountered, control passes to the statement immediately following the `switch` body.

### Example

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

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

## Iteration Statements

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

C has several iteration statements: `while`, `do-while`, and `for`. The main difference between these statements is the point at which each loop tests for the exit condition. Refer to the `goto`, `continue`, and `break` statements for ways to exit a loop without reaching its end or meeting its exit condition.

### Syntax

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

### Examples

These three loops all accomplish the same thing (they assign `i` to `a[i]` for i from 0 to 4):

#### The while loop

```
i = 0;
while (i < 5)
{
   a[i] = i;
   i++;
}
```

#### The do-while loop

```
i = 0;
do
{
   a[i] = i;
   i++;
} while (i < 5);
```

#### The for loop

```
for (i = 0; i < 5; i++)
{
   a[i] = i;
}
```

## The while Statement

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

### Syntax

```
while (expression)
    statement
```

### Description

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

### Example

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

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

## The do Statement

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

### Syntax

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

### Description

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

### Example

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

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

## The for Statement

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

### Syntax

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

### Description

The `for` statement is a general-purpose looping construct that allows you to specify the initialization, termination, and increment of the loop. The `for` uses three expressions. Semicolons separate the expressions. Each expression is optional, but you must include the semicolons.

*`Expression1`* is the *initialization expression* that typically specifies the initial values of variables. It is evaluated only once before the first iteration of the loop.

*`Expression2`* is the *controlling expression* that determines whether or not to terminate the loop. It is evaluated before each iteration of the loop. If *`expression2`* evaluates to a nonzero value, the loop body is executed. If it evaluates to 0, execution of the loop body is terminated and control passes to the first statement after the loop body. This means that if the initial value of *`expression2`* evaluates to zero, the loop body is never executed.

*`Expression3`* is the *increment expression* that typically increments the variables initialized in *`expression1`*. It is evaluated after each iteration of the loop body and before the next evaluation of the controlling expression.

The `for` loop continues to execute until *`expression2`* evaluates to 0, or until a jump statement, such as a `break` or `goto`, interrupts loop execution.

If the loop body executes a `continue` statement, control passes to *`expression3`*. Except for the special processing of the `continue` statement, the `for` statement is equivalent to the following:

```
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 example:

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

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

## Jump Statements

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

### Syntax

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

### Examples

These four fragments all accomplish the same thing (they print out the multiples of 5 between 1 and 100):

```
   i = 0;
   while (i < 100)
   {
      if (++i % 5)
         continue;   /* unconditional jump to top of while loop */
      printf ("%2d ", i);
   }
   printf ("\n");


   i = 0;
L: while (i < 100)
   {
      if (++i % 5)
         goto L:     /* unconditional jump to top of while loop */
      printf ("%2d ", i);
   }
   printf ("\n");


   i = 0;
   while (1)
   {
      if ((++i % 5) == 0)
         printf ("%2d ", i);
      if (i > 100)
         break;      /* unconditional jump past the while loop  */
   }
   printf ("\n");


   i = 0;
   while (1)
```

```
{
   if ((++i % 5) == 0)
      printf ("%2d ", i);
   if (i > 100) {
      printf ("\n");
      return;     /* unconditional jump to calling function  */
   }
}
```

## The goto Statement

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

### Syntax

```
goto identifier ;
```

### Description

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

## The continue Statement

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

### Syntax

```
continue;
```

### Description

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

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

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

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

### Example

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

This example prints:

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

## The break Statement

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

### Syntax

```
break;
```

### Description

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

When used in the `switch` statement, `break` normally terminates each `case` statement. If you use no `break` (or other unconditional transfer of control), each statement labeled with `case` flows into the next. Although not required, a `break` is usually placed at the end of the last case statement. This reduces the possibility of errors when inserting additional cases at a later time.

For example:

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

This example prints:

```
     0
     1
     2
```

## The return Statement

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

### Syntax

```
return [expression];
```

### Description

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

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

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

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

# 7 Preprocessing Directives

Preprocessing directives function as compiler control lines. They enable you to direct the compiler to perform certain actions on the source file. You can use the preprocessing directives to make a number of textual changes in the source before it is syntactically and semantically analyzed and translated. Since preprocessing occurs conceptually before the compilation process, there is generally no relationship between the syntax of a translation unit and preprocessing directives. There are some restrictions on where `#pragma` directives may appear within a translation unit. Refer to chapter 8 for details.

### Syntax

```
preprocessor-directive :=
    include-directive newline
    macro-directive newline
    conditional-directive newline
    line-directive newline
    error-directive newline
    pragma-directive newline
```

### Description

The preprocessing directives control the following general functions:

1. **Source File Inclusion**

   You can direct the compiler to include other source files at a given point. This is normally used to centralize declarations or to access standard system headers such as `stdio.h.sys`.

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 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. 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 (\).

**NOTE** In ANSI mode, white space may precede the # character in preprocessing directives.

Comments in the source file that are not passed by default through the preprocessor are replaced with a single white-space character. **Examples**

*include-directive*: `#include <stdio.h>`

*macro-directive*: `#define MAC x+y`

*conditional-directive*: `#ifdef MAC`

*line-directive*: `#line 5 "myfile"`

*pragma-directive*: `#pragma INTRINSIC func`

## Source File Inclusion

You can include the contents of other files within the source file using the `#include` directive.

### Syntax

```
include-directive :=
    #include <filename>
    #include "filename"
    #include identifier
```

### Description

The `#include` preprocessor directive enables you to insert the contents of the specified external file into the source file prior to compilation. The file name in the `#include` directive may be enclosed in angle brackets (< >) or double quotation marks . File names enclosed in angle brackets are assumed to be standard include files that are provided with the HP C/iX compiler. All standard include files reside in the H group of the SYS account.

The arguments to the `#include` directive are subject to macro replacement before the directive processes them. In the third form above, *`identifier`* must be in the form of one of the first two choices after macro replacement by the preprocessor.

For example:

```
#define varname "my_file"
#include varname
```

Error messages produced by the compiler usually supply the file name where the error occurred as well as the file relative line number of the error.

The HP C/iX preprocessing phase allows for the use of non-standard (UNIX [1] -like root names) file names in certain `#include` directives. This minimizes the required source code changes when transporting code between different systems.

The preprocessor strips include file names enclosed in angle brackets of all prefixes and suffixes. The preprocessor then searches for a file with the resulting name in the `H` group of the `SYS` account. If the file is not found, an error is issued.

For example, if you specify the following directive:

```
#include <stdio.h>
```

the preprocessor searches for `STDIO.H.SYS`. If you specify:

```
#include <sys/errno.h>
```

the preprocessor strips the `sys/` prefix and searches for `ERRNO.H.SYS`.

1. UNIX is a trademark of AT&T in the U.S. and other countries.

## Examples

```
#include <stdio.h>

#include "myheader"

#ifdef   MINE
#   define  filename  "file1"
#else
#   define  filename  "file2"
#endif

#include filename
```

## Preprocessor Search of Include Files in Quotation Marks

The preprocessor searches for include files enclosed in double quotation marks, (for example `#include myfile`), as follows:

1. If the include *`filename`* contains an MPE/iX group and account name, then the preprocessor searches the specified group and account. If no group and account is specified, it searches the group and account where the source file resides. If *`filename`* is not found, the preprocessor performs step 2.
2. The preprocessor removes any extensions and adds the `H` group to *`filename`*. It then searches the `H` group of the account where the source file resides. If the preprocessor does not find *`filename`*, it performs step 3.
3. The preprocessor adds the `SYS` account to *`filename`*`.H` and searches for *`filename`*`.H.SYS`.
4. If the *`filename`* is still not found, the preprocessor issues an error.

## Examples

```
#include "MYFILE"
```

The preprocessor starts by looking for `MYFILE` in the group and account where the source file is located. It then looks for `MYFILE.H` in the account where the source file is located, and finally looks for `MYFILE.H.SYS`.

```
#include "MYFILE.X"
```

Assuming the source file is located in `SRCGROUP.SRCACCT`, the preprocessor looks for the include file in the following order:

1. `MYFILE.X.SRCACCT`
2. `MYFILE.SRCGROUP.SRCACCT`
3. `MYFILE.H.SRCACCT`
4. `MYFILE.H.SYS`

## Include Files Within Include Files

Files that are included can also use the `#include` directive. The compiler supports up to 35 levels of nested include files.

Note that this search path is based on the group and account where the source file is located and not where the include file is "included". To illustrate, assuming that a program MAIN.SRCGROUP.SRCACCT contains the following directive:

```
#include "HEADER1.X.Y"
```

and HEADER1.X.Y contains another include directive:

```
#include "HEADER2"
```

The preprocessor looks for HEADER2.SRCGROUP.SRCACCT even though the file that includes it is in X.Y.

The above searching algorithm can be modified in 2 ways; by using file equations or by using the -I option. Since include files are always fully qualified before searching by the preprocessor, a fully qualified file name must be used in a file equation to be effective.

## Example

```
:file MYFILE.SRCGROUP.SRCACCT=HEADER.X.Y
#include "MYFILE"
```

This directs the preprocessor to look for HEADER.X.Y instead of MYFILE.SRCGROUP.SRCACCT.

## -I Compiler Option

You can alter the search algorithm used for locating included files by specifying the -I compiler option in the following format:

-I*group*[.*account*]

You use this option to specify a group and optionally an account where the preprocessor searches for included files before searching the source file's H group and the H group of the SYS account. If you omit the account specification, the account of the current user is used. Note that this is the *only* case where the user's account is used when searching for the include files. You can cause the preprocessor to search more than one group (and account) for included files by specifying more than one -I option. -I options are scanned left to right.

If the included file is enclosed in angle brackets, the preprocessor strips the name of non-standard prefixes and suffixes and then searches the groups and accounts in the -I option(s) for the file. If the file is not found, the H group of the SYS account is searched.

If the included file name is enclosed in double quotation marks, the preprocessor first searches for that file without changing the file name. It then strips the file name of prefixes and suffixes and searches in the group and the account of the source file. If the file is not found, the groups and accounts, if any, specified in the -I option(s) are searched. Then, the H group of the source file is searched. If the file is still not found, the H group of the SYS account is searched last.

In summary, the preprocessor searches for names of included files enclosed in angle brackets only in the groups or accounts specified in any -I options, then in the H group of the SYS account, but never searches the group and account in which the source file is located. When the name is enclosed in double quotation marks, the group and account in which the source file is located is searched. If you use the `-I` compiler option, the indicated groups and accounts are searched *before* the H and H.SYS groups and accounts are searched, regardless of which characters surround the name.

## Example

If you are compiling a local file `TESTFILE` in GRP1.ACCT1 that includes the preprocessor directive:

```
#include "MYHEADER"
```

and you want it to include the file `MYHEADER` in `HOMEGRP.HOMEACCT`, use the command:

```
CCXL TESTFILE;INFO="-IHOMEGRP.HOMEACCT"
```

## Recommendations for Using Include Files

If you have include files or source files spread throughout your file system, you may want to become familiar with the following recommendations for arranging include files in a flexible and consistent manner. However, if you keep all source and include files in a single group and account on your system, the recommendations listed below may not be useful to you.

There is no best way to arrange include files on a system but there are practices to follow that make the use of include files simple and efficient for most cases. Include file searching for standard files enclosed by angle brackets is fully described in chapter 8. The following recommendations are for include file names enclosed by double quotes.

- Put include files in the same group and account as the source file with which they are associated. This is the default search place for files that are not fully qualified and for qualified files that are not found in the first try.
- If you have *standard* files to be shared by source files in the same account but in different groups, put them in the H group of the source file's account. Be aware that if source files specify `file.h` for these standard files, a FILE equation command is required to open any file other than `file.h.srcacct`. Consequently, this is the first file opened by the preprocessor because the account of the source file is implied before any -I options are examined.
- Use the `-I` option to tell the preprocessor where to find include files that are not in the exact group or the H group of the source file's account. Qualify the `-I` option with group and account if you want searches in the account of the source file. Omit the account specification only if you want searches made in the account of the current user, as you might when using job files that log on in other accounts.
- If you want to override the defaults for a single include file, or small subset of include files, use MPE/iX's FILE command to equate the file that is opened by the preprocessor to the file that you want to use. Use the rules for include file searching to determine exactly what file the preprocessor opens and use that fully qualified name in the file equation.

### Examples

For the source file `FILE1.MYGRP.OURACCT` that needs a special header file called `MYDEFS`, put `MYDEFS` in `MYGRP.OURACCT` and use:

```
#include "MYDEFS"
```

in `FILE1.MYGRP.OURACCT` . The name `MYDEFS.H` in the include statement also works, as does putting the `MYDEFS` file in `H.OURACCT`, but neither of those choices reflects the intended use of the file `MYDEFS`.

If you want to substitute the file `FIXDEFS` for `MYDEFS` when compiling `FILE1.MYGRP.OURACCT` in the above example, use the file equation:

```
:FILE MYDEFS.MYGRP.OURACCT = FIXDEFS
```

It is important to note that a fully qualified formal designator is required for the file to be successfully equated, as the preprocessor builds a fully qualified file name before attempting an open. The exception to this rule is when the preprocessor opens a file using a -I option that only specifies a group. If a file equation and this type of -I option are both needed to find a file, it is likely that the file could be moved to a more effective location and a simpler searching strategy used.

For the source file `FILE2.MYGRP.OURACCT` that needs a general header file called `OURDEFS`, put `OURDEFS` in `H.OURACCT` and use:

```
#include "OURDEFS.H"
```

or use:

```
#include "OURDEFS"
```

in `FILE2.MYGRP.OURACCT` . The first form is more restrictive because it finds the file before the `-I` options are examined. The second allows a `-Igrp.acct` specification to be used to include a different set of header files.

In summary, the best overall strategy is to locate your include files in the appropriate groups and accounts and use FILE commands and the `-I` compiler option to handle exceptional cases.

## Macro Replacement

You can define text substitutions in your source file with C macro definitions.

### Syntax

```
macro-directive :=
    #define identifier [replacement-list]
    #define identifier ( [identifier-list] )
         [replacement-list]
    #undef identifier

replacement-list :=
    token
    replacement-list token
```

### Description

A `#define` preprocessing directive of the form:

`#define identifier [replacement-list]`

defines the *identifier* as a macro name that represents the replacement list. The macro name is then replaced by the list of tokens wherever it appears in the source file (except inside of a string or character constant, or comment). A macro definition remains in force until it is undefined through the use of the `#undef` directive or until the end of the translation unit.

Macros can be redefined without an intervening `#undef` directive. Any parameters used must agree in number and spelling, and the replacement lists must be identical. All whitespace 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 whitespace. 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

ANSI C provides the , , , , and  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 |
|---|---|
| | Produces the date of compilation in the form `Mmm dd yyyy`. |
| | Produces the name of the file being compiled. |
| | Produces the current source line number. |
| | Produces the decimal constant 1, indicating that the implementation is standard-conforming. |
| | Produces the time of compilation in the form `hh:mm:ss`. |

**NOTE** , , and  are only defined in ANSI mode.

In addition to the above macros, HP C/iX provides the following predefined macros, which can be used when cross-compiling between the HP-UX and MPE/iX operating systems:

## Conditional Compilation

Conditional compilation directives allow you to delimit portions of code that are compiled if a condition is true.

### Syntax

```
conditional-directive :=
    #if     constant-expression newline [group]
    #ifdef  identifier newline [group]
    #ifndef identifier newline [group]
    #else newline [group]
    #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 are considered true if the *identifier* was previously defined using a #define directive.

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

The following are examples of valid combinations of these conditional compilation directives:

```
#ifdef SWITCH
                 /* compiled if SWITCH is defined */
#else
                 /* compiled if SWITCH is undefined */
#endif                /* end of if */

#if defined(THING)
                 /* compiled if THING is defined */
#endif                /* end of if */

#if A>47
                 /* compiled if A 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 */

#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           */
                         /* SMALL_MODEL are defined, INT_SIZE    */
                         /* is defined to be 16 (bits).          */

#ifdef DEBUG                     /* If DEBUG is defined, display */
   printf("table element : \n"); /* the table elements.          */
   for (i=0; i < MAX_TABLE_SIZE; ++i)
      printf("%d  %f\n", i, table[i]);
#endif
```

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

## Line Control

You can cause the compiler to increment line numbers during compilation from a number specified in a line control directive. (The resulting line numbers appear in error message references, but do not alter the line numbers of the actual source code.)

### Syntax

```
line-directive :=
  #line digit-sequence [filename]
```

### Description

The `#line` preprocessing directive causes the compiler to treat lines following it in the program as if the name of the source file were *filename* and the current line number is *digit-sequence*. This is to control the file name and line number that is given in diagnostic messages, for example. This feature is used primarily for preprocessor programs that generate C code. It enables them to force the 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.

The compiler defines two macros that you can use for error diagnostics. The first is , an integer constant equal to the value of the current line number. The second is , a quoted string literal equal to the name of the input source file. Note that you can change  and using `#include` or `#line` directives.

### Example

```
#line digit-sequence [filename]:  #line 5 "myfile"
```

## Pragma Directive

You can provide instructions to the compiler through inclusion of pragmas.

### Syntax

```
pragma-directive :=
#pragma replacement-list
```

### Description

The `#pragma` preprocessing directive provides implementation-dependent information to the compiler. Any pragma that is not recognized by the compiler is ignored.

See chapter 8 for descriptions of pragmas recognized by HP C/iX.

### Example

```
  #pragma replacement-list:  #pragma intrinsic func
```

## Error Directive

### Syntax

```
#error [pp-tokens]
```

The `#error` directive causes a diagnostic message, along with any included token arguments, to be produced by the compiler.

### Examples

```
#ifndef (HP_C)
#error "HP_C not defined!"      /* This directive will produce  */
#endif                          /* diagnostic message "HP_C not */
                                /* defined!"                    */

#if TABLE_SIZE % 256 != 0
#error "TABLE_SIZE must be a multiple of 256!"
#endif                          /* This directive will produce  */
                                /* the diagnostic message       */
                                /* "TABLE_SIZE must be a        */
                                /* multiple of 256!" */
```

**NOTE** The `#error` directive is only supported in ANSI mode.

## Trigraph Sequences

The C source code character set is a superset of the ISO 646-1983 Invariant Code Set. To enable programs to be represented in the reduced set, *trigraph sequences* are defined to represent those characters not in the reduced set. A *trigraph* is a three character sequence that is replaced by a corresponding single character. Table 7-2 gives the complete list of trigraph sequences and their replacement characters.

**Table 7-2. Trigraph Sequences and Replacement Characters**

| **Trigraph Sequence** | **Replacement** |
|---|---|
| ?? | # |
| ??/ | |
| ??' | ^ |
| ??( | [ |
| ??) | ] |
| ??! | \| |
| ?? | { |
| ?? | } |
| ??- | ~ |

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

**NOTE** Trigraphs are replaced in ANSI mode only.

# 8 Compiling and Running HP C/iX Programs

This chapter describes how to compile, link, and run HP C programs on the MPE/iX operating system. The following steps must occur before you can execute an HP C/iX program:

1. Translate the source code into an object file.
2. Link one or more object files into a program file.
3. Load and execute the program file.

You can perform each of these steps independently, controlling the details of each step. Use the MPE/iX commands CCXL, LINK, and RUN to accomplish steps 1, 2, and 3, respectively.

Alternatively, you can combine the steps using a single MPE/iX command. CCXLLK performs steps 1 and 2, while CCXLGO performs steps 1, 2, and 3.

## Compiling HP C/iX Programs

You can compile HP C/iX programs using the MPE/iX commands CCXL, CCXLLK or CCXLGO, or by explicitly running the CCOMXL.PUB.SYS program.

### CCXL Command

The CCXL command invokes the HP C/iX compiler and generates an object file.

#### Syntax

```
CCXL [textfile] [,[objectfile] [,[listfile]] [;INFO="options"]
```

#### Parameters

| | |
|---|---|
| *textfile* | is the source file that the HP C/iX compiler reads. If omitted, the default is `$STDIN`. |
| *objectfile* | is the relocatable object file to which the compiler writes the object code. If omitted, the default is `$NEWPASS`. |
| *listfile* | is the listing file. If omitted, the default is `$STDLIST`. |
| `options` | are compiler options you want to take effect; separate options with a blank. See "HP C/iX Compiler Options" later in this chapter for specific options. |

#### Description

If you omit *textfile*, the current input device, `$STDIN`, is used by default. Typically, the terminal is the standard input device, and this allows you to enter source code interactively. Indicate the end of the interactive session by entering a colon (`:`).

If you omit *listfile*, the standard listing file, `$STDLIST`, is used by default. Typically, a listing is sent to the terminal during a terminal session or to the printer in a batch job. If *listfile* is a file other than `$STDLIST`, the compiler writes errors and warnings to `$STDLIST` and *listfile*.

#### Examples

```
  CCXL MYTEXT,,MYLIST
```

This example compiles the HP C/iX source file `MYTEXT`, puts the object code in `$NEWPASS` (by default), and writes the list file to `MYLIST`.

```
  CCXL MYTEXT,MYOBJ;INFO="-Ddebug -v"
```

This example compiles the source file `MYTEXT`, places the object code in the file `MYOBJ`, sends the list file to the terminal, and passes two options to the compiler. The `-Ddebug` option defines `debug` as if it were defined using the `#define` preprocessor statement and has the value of 1. The `-v` option echoes the different stages of processing the source file goes through during compilation.

## CCXLLK Command

The CCXLLK command invokes the HP C/iX compiler, generates an object file, and links the object file with the HP C/iX library to produce an executable program file.

### Syntax

```
CCXLLK [textfile] [,[programfile] [,[listfile]] [;INFO="options"]
```

### Parameters

| | |
|---|---|
| *textfile* | is the source file that the HP C/iX compiler reads. If omitted, the default is `$STDIN`. |
| *programfile* | is the program file to which the linker writes the linked program. If omitted, the default is `$NEWPASS`. |
| *listfile* | is the listing file. If omitted, the default is `$STDLIST`. |
| `options` | are compiler options you want to take effect; separate options with a blank. See "HP C/iX Compiler Options" later in this chapter for specific options. |

### Description

If you omit *`textfile`*, the current input device, `$STDIN`, is used by default. Typically, the terminal is the standard input device, and this allows you to enter source code interactively. You should indicate the end of the interactive session by entering a colon (`:`).

If you omit *`listfile`*, the standard listing file, `$STDLIST`, is used by default. Typically, a listing is sent to the terminal during a terminal session or to the printer in a batch job. If *`listfile`* is a file other than `$STDLIST`, the compiler writes errors and warnings to `$STDLIST` *and* *`listfile`*.

### Examples

```
  CCXLLK MYTEXT,MYPROG
```

This example compiles the source file `MYTEXT`, places the linked program in `MYPROG`, and writes the list file to `$STDLIST`.

```
  CCXLLK ,MYPROG,MYLIST;INFO="-Wc,-r"
```

This example compiles from `$STDIN` (by default), places the linked program in `MYPROG`, writes the list file to `MYLIST`, and passes the `-Wc,-r` option to the compiler.

## CCXLGO Command

The CCXLGO command invokes the HP C/iX compiler, generates an object file, links the object file with the HP C/iX library to produce an executable program, and then runs the program.

### Syntax

```
CCXLGO [textfile] [,[listfile] [;INFO="options"]
```

### Parameters

| | |
|---|---|
| *textfile* | is the source file that the HP C/iX compiler reads. If omitted, the default is `$STDIN`. |
| *listfile* | is the listing file. If omitted, the default is `$STDLIST`. |
| `options` | are compiler options you want to take effect; separate options with a blank. See "HP C/iX Compiler Options" later in this chapter for specific options. |

### Description

If you omit *textfile,* the current input device, $STDIN, is used by default. Typically, the terminal is the standard input device, and this allows you to enter source code interactively. You should indicate the end of the interactive session by entering a colon (:).

If you omit *listfile,* the standard listing file, $STDLIST, is used by default. Typically, a listing is sent to the terminal during a terminal session or to the printer in a batch job. If *listfile* is a file other than $STDLIST, the compiler writes errors and warnings to $STDLIST *and listfile*.

CCXLGO has the side effect of creating a temporary program file named $NEWPASS.

### Example

```
CCXLGO MYTEXT,$NULL
```

This example compiles the HP C/iX source file MYTEXT without listing it to the terminal, links and runs the resulting program, and places the linked program in $NEWPASS. If $NULL is omitted, the compilation listing appears on the screen.

## RUN CCOMXL.PUB.SYS

You can also compile HP C/iX programs using the MPE RUN command. The HP C/iX compiler is located in the program file CCOMXL in the PUB group of the SYS account. The compiler uses the default files unless you override the default values. To override the default values you needed to perform the following steps:

1. Equate the file you want to substitute for the default file with its formal file designator using the MPE FILE command.
2. Select an appropriate value of the PARM parameter of the RUN command. This value indicates which files are not defaulted.

The HP C/iX compiler recognizes the following default files and formal file designators:

**Table 8-1. Default Files and Designators**

| File Type | Default | Designator |
|---|---|---|
| Source | $STDIN | CCTEXT |
| Object | $NEWPASS | CCOBJ |

**Table 8-1. Default Files and Designators**

| File Type | Default | Designator |
|---|---|---|
| List | `$STDLIST` | `CCLIST` |

The PARM parameter of the RUN command indicates which files have been equated. This directs the compiler to use these files instead of the default files. The RUN command takes a PARM parameter with an integral value in the range 0 to 7. The low-order three bits of the PARM value represent the source, object, and list files as shown in the following diagram.

```
           Bit 29       Bit 30       Bit 31

       |   object    | listing   |   source    |
```

The integral value sets the low-order bits as shown in the following table.

**Table 8-2. Low-order Bit Values**

| Value | Files in FILE Commands |
|---|---|
| 0 | None |
| 1 | Source |
| 2 | Listing |
| 3 | Listing, source |
| 4 | Object |
| 5 | Object, source |
| 6 | Object, listing |
| 7 | Object, listing, source |

An error occurs if you use a PARM value that sets a bit for a file for which no file equation exists. If a file equation exists, but the bit is not set in the PARM value, the compiler uses the default value.

The RUN command also has an optional INFO parameter. You can use this parameter to pass options, delimited with blanks, to the compiler.

## Example

```
FILE CCTEXT = MYFILE
FILE CCLIST = MYLIST
RUN CCOMXL.PUB.SYS;PARM=3;INFO="-O"
```

This example compiles the file `MYFILE`, places the object code in `$NEWPASS` (the default), writes the listing to `MYLIST`, and passes the `-O` option to the compiler.

## HP C/iX Compiler Options

You can pass options to the HP C/iX compiler in the INFO parameter of the CCXL, CCXLLK, and CCXLGO commands, or using the INFO parameter of the RUN command, if you invoke the compiler using CCOMXL.PUB.SYS. You must separate options with a blank. Any string of characters not separated with a blank is considered a single option. For example, `-Wc,-r` is considered one option with `-r` as an argument to the option; it inhibits the promotion of `float` to `double`. In addition, note that the case of the options *is* significant.

The following options are available:

| **Option** | **Description** |
|---|---|
| `-A level` | where `level` can be `a` or `c`. |
| `a` | Requests a compilation on ANSI C mode. By using the `-Aa` option, you are requesting a strict implementation of ANSI C. ANSI C specifies which names are available in the standard libraries and headers, which are reserved for the implementation, and which must be left available for the user. HP C/iX in ANSI mode conforms to these restrictions, and only names permitted by ANSI C are defined or declared in the standard libraries and headers. However, if you want to compile using ANSI mode but want the naming restriction relaxed, you can include the following line in the start of your source before including any header files:<br>`#define _MPEXL_SOURCE`<br>This will allow you to gain access to names that are legal in non-ANSI mode. |
| `c` | This is a non-ANSI implementation and the default compilation mode. |
| `-C` | Prevents the preprocessing phase from stripping C comments. |
| `-D`*`name`*<br>`-d`*`[name=def]`* | Defines *`name`* to the preprocessing phase as if defined using the `#define` directive. If the definition is not given, *`name`* is defined as `1`. |
| `-E` | Only runs the preprocessing phase on the input file and sends the result to `$STDLIST`. When you use this option, object file and listing file specifications are ignored. |
| `-g` | Causes the compiler to generate additional information needed by the Symbolic Debugger. This option is incompatible with optimization. |
| `-I`*`group[.acct]`* | Changes the search algorithm used by the preprocessing phase for finding `#include` files. For additional information, see chapter 7. |
| `-O` | Invokes the optimizer to perform all optimizations. This option is not compatible with symbolic debugging. Refer to the *HP C Programmer's Guide* for details on optimization. |

| **Option** | **Description** |
|---|---|
| `-P` | Only runs the preprocessing phase on the source file and stores the result in the file normally used for the object file. For example, preprocessor output from the command `CCXL MYSOURCE, CPPLIST;INFO="-P"` is sent to the ASCII file `CPPLIST`. |
| `-U`*`name`* | Removes any initial definition of *`name`* in the preprocessing phase. |
| `-v` | Enables the verbose mode, producing a step-by-step description of the compilation process on the listing file. |
| `-w` | Suppresses warning messages. |
| `-W`*`x,arg1[, arg2, ...,argn]`* | Hands off the arguments *`arg1`* through *`argn`* to the phase *`x`* of the compilation; *`x`* can be one of the following values: |

| **Value** | **Description** |
|---|---|
| `p` | Preprocessor |
| `c` | Compiler |

The *`+arg1 +arg2`* notation can be used as a shorthand for the *`-arg1, -arg2`* notation.

The arguments to the compiler option can be one or more of the following:

| **Argument** | **Description** |
|---|---|
| `-C`*`size`* | Creates the output file for the preprocessing phase with a record length of *`size.`* The default is 512 bytes. |
| `-e` | Allows the use of extension features, such as long pointers and using the $ character in the identifier name. |
| `-F`*`size`* | Creates the output file for the preprocessing phase with a limit of *`size.`* The default is 7500 records. |
| `-m` | Causes the identifier maps to be printed. Refer to chapter 11, "The Listing Facility," for listing options and format. |
| `-o` | Causes the code offsets to be printed. Refer to chapter 11, "The Listing Facility," for listing options and format. |
| `-Obb`*`num`* | Specifies the maximum number of basic blocks allowed in a procedure which is to be optimized at level 2. A basic block is a sequence of code with a single entry point, single exit point, and no internal branches. Optimizing procedures with a large number of basic blocks can take a long time and use a large amount of memory. If the limit is exceeded, a warning message lists the name of the procedure and the number of basic blocks it contains, and then level 1 optimization is performed. The default value for this limit, if this option is not present is 500. This option implies level 2 optimization (equivalent to `-O` or +02). |

| **Argument** | **Description** |
|---|---|
| `-O`*`opt`* | Invokes optimizations selected by *`opt`*. If *`opt`* is `1`, only level 1 optimizations are performed. If *`opt`* is 2, all optimizations are performed. The option `+O2` is the same as `-O`. |
| `-R`*`num`* | Only allows the first *`num`* 'register' variables to actually have the 'register' class. Use this option when the register allocator issues an "out of general registers" message. |
| `-r` | Inhibits the automatic promotion of `float` to `double` in evaluating expressions and passing arguments (`-r` is invalid in ANSI mode). |
| `-u` | Forces the compiler to generate code to access pointers with half-word addressing. You should only use this option when you cannot guarantee that pointers will always reference word-aligned items. See "Pointers to Half-Word Aligned Data Items" in chapter 9 for more information. |
| `-w`*`n`* | Specifies the level of the warning messages; *`n`* can be one of the following values: |

| **Value** | **Description** |
|---|---|
| 1 | All warnings are issued. |
| 2 | Only warnings indicating that code generation might be affected are issued; equivalent to the compiler default without any *`w`* options. |
| 3 | No warnings are issued. |

The preprocessor phase supports the aforementioned `-C, -D`, `-I`, and `-U` options, as well as the following:

| | |
|---|---|
| `-H`*`n`* | Changes the internal macro definition table to be *`n`* bytes in size. The macro symbol table is increased proportionally. You should specify a value greater than the 128000 byte default. Use this option when the preprocessing phase issues a "too many defines" or "too much defining" message. |
| `-P` | Processes the input without producing the line control information used by the next pass of the compiler. |
| `-T` | Forces the preprocessor to use only the first eight characters in distinguishing different preprocessor names (included for backward compatibility to other systems). |

## Examples

```
CCXL MYTEXT;INFO="-Ddebug -Wc,-r -O"
```

This example compiles `MYTEXT` with `debug` defined as 1 (`-Ddebug`), inhibits promotion of float expressions to double (`-Wc,-r`), and enables all possible optimizations (`-O`).

```
CCXL MYTEXT,CPPOUT;INFO="-P -C -Ddebug"
```

This example only executes the preprocessing phase on MYTEXT(-P), leaves the output with the comments intact in CPPOUT (-C), and defines debug as 1 (-Ddebug).

## CCOPTS CI Variable

Options may also be passed to the compiler using the Command Interpreter variable CCOPTS. The compiler picks up the value of CCOPTS and places its contents before any arguments in the INFO string.

### Example

```
SETVAR CCOPTS "-g"
CCXL MYFILE;INFO="-v"
```

is equivalent to:

```
CCXL MYFILE;INFO="-g -v"
```

## Pragmas

You may include the following pragmas within a source file, but you may not use them within a function. A pragma is valid from the point that it is included to the end of the source file or until another pragma changes its status.

[#pragma OPTIMIZE {ONOFF }]

Turns all optimizations ON or OFF, depending on which option you use.

[#pragma OPT_LEVEL {12 }]

Sets optimization level to local when you specify `OPT_LEVEL 1`, or sets optimization level to global *and* local when you specify `OPT_LEVEL 2`.

`#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 information allows better optimizations to be performed when optimization level 2 has been specified.

`#pragma ALLOCS_NEW_MEMORY` *functionname*$_1$*, . . . ,functionname*$_n$

States that the function *functionname* returns a pointer to "new" memory, such as heap space, that it allocates or a routine that it calls allocates. This information allows better optimizations to be performed when optimization level 2 has been specified.

`#pragma COPYRIGHT` *"string"*

Places a copyright notice using *string* as the company name into the relocatable object module or the program file.

`#pragma COPYRIGHT_DATE` *"string"*

Specifies a date string to be used in a copyright notice appearing in an object module.

`#pragma VERSIONID` *"string"*

Specifies a version number to be associated with a particular piece of software. The *string* is placed into the object file produced when the software is compiled.

`#pragma PAGE`

Causes a page break in the listing and begins a new page.

`#pragma LINES` *linenum*

Sets the number of lines per page to *linenum.*

`#pragma WIDTH` *pagewidth*

Sets the width of the page to *pagewidth.*

`#pragma TITLE` *"string"*

Makes *string* the title of the listing.

`#pragma SUBTITLE` *"string"*

Makes *string* the subtitle of the listing.

[#pragma LIST {ONOFF }]

Turns listing functionality ON or OFF. The default is ON.

[#pragma AUTOPAGE {ONOFF }]

When ON, causes a page break in the listing after each function definition. The default is OFF.

`#pragma LOCALITY` *`"string"`*

Specifies a name to be associated with the code that is written to a Relocatable Object Module. All code following the locality pragma will be associated with the name given until the end of the current source file or until another locality pragma is encountered. Locality of code is at the function level. Without the locality pragma, the name CODE is associated with the generated code.

For more information on optimizer pragmas, refer to chapter 7. For more information on the listing pragmas, refer to chapter 11.

[#pragma HP_ALIGN {MPE_16POP}]

The `MPE_16` option directs the HP C/iX compiler to set the alignment of `int`, `float`, and `double` in structures and unions to be aligned according to the MPE/V alignment scheme. This option also sets the alignment of structures and unions to start and end on at least a half-word boundary. The `HP_ALIGN MPE_16` pragma facilitates reading TurboImage databases and MPE/V based binary files on MPE/iX systems.

The `POP` option turns off the `HP_ALIGN` pragma and alignment reverts to word (32-bit) alignment. For more information on data alignment, refer to chapter 9, "HP C/iX Implementation Topics."

## Linking the C Library

This section describes the procedure necessary to link C library functions into your program using the MPE/iX LINK command. For many applications, linking the LIBCINIT.LIB.SYS file in the RL list, as done by the CCXLLK and CCXLGO commands, is sufficient. Applications that use mathematical or random number functions, or programs compiled using ANSI mode, will nee to link with additional libraries.

### C Library Organization

The C library consists of several files that may be separated into two functional areas: the standard library and the mathematical library.

The standard library consists of the input/output functions, the general utility functions that perform operations such as string and memory manipulation, and the program startup functions. All C programs must link in the standard library because it contains the startup routines necessary for program execution. Failure to link in this library will result in a linker or loader error. The standard library is available in the system executable library (XL.PUB.SYS) and also as a relocatable library (RL).

The math library consists of additional mathematical functions, such as the trigonometric and logarithmic functions, that perform floating point operations. The math library is only available in RL form.

For further information on the organization of the HP C/iX Library, refer to the *HP C/iX Library Reference Manual*.

### Linking the Library Files

To use the executable standard library, add the LIBCINIT.LIB.SYS file to the RL list when linking your program. To use the relocatable standard library, add the LIBC.LIB.SYS file to the RL list when linking. Note that you may choose either the executable (XL) or relocatable (RL) library form, but not both.

Additionally, if the program is compiled using ANSI mode (the `-Aa` option), the relocatable library `LIBCANSI.LIB.SYS` must also be added to the RL list regardless of whether `LIBCINIT.LIB.SYS` or `LIBC.LIB.SYS` is used. This is to ensure that certain file behaviors conform to ANSI specifications. For a detailed description of `LIBCANSI.LIB.SYS`, see the *HP C/iX Library Reference Manual*. For a detailed discussion of the LINK command, see the *MPE/iX Commands Reference Manual*.

To use the math library, add the LIBM.LIB.SYS file to the RL list when linking your program. The LIBM.LIB.SYS file must precede the LIBC.LIB.SYS file in the RL list if the RL form of the standard library is used. The ordering of the files is significant because of the interdependencies of the libraries. The ordering is not significant if the XL form of the standard library is linked.

In addition to `LIBM.LIB.SYS`, there is also `LIBMANSI.LIB.SYS`, the ANSI conforming version of the math library. You must decide which version of the math library to use and link with the appropriate RL.

The `rand` and `srand` functions are conceptually part of the standard library but reside in a different library file, LIBCRAND.LIB.SYS. To use these functions, add the LIBCRAND.LIB.SYS file to the RL list when linking your program. These functions are not available in XL form. This special treatment for the `rand` and `srand` functions is due to a name conflict between the HP C/iX library function `rand` and the MPE/iX compiler library function `rand`.

### Examples

To link the object file `MYOBJ` and the RL form of the standard library into the program file `MYPROG`, enter:

```
LINK FROM=MYOBJ; TO=MYPROG; RL=LIBC.LIB.SYS
```

To link the object file `MYOBJ` together with the random number generation functions `rand` and `srand`, the math library routines, and the RL form of the standard library, enter:

```
LINK FROM=MYOBJ; TO=MYPROG; RL=LIBCRAND.LIB.SYS,LIBM.LIB.SYS,LIBC.LIB.SYS
```

Remember that either `LIBC.LIB.SYS` or `LIBCINIT.LIB.SYS` must be linked into your program.

To link a program under ANSI mode, enter:

```
LINK FROM=OBJFILE; TO=PROGFILE; RL=LIBCINIT.LIB.SYS,LIBCANSI.LIB.SYS
```

Non-ANSI is the default mode.

## Running HP C/iX Programs

You can run HP C/iX programs using the RUN command or by entering the program name (an implied run). You can pass parameters to the main program and redirect the standard input (`stdin`), standard output (`stdout`), and error output (`stderr`) to specific files by using the `INFO` string.

### Program Parameters

You can pass parameters to an HP C/iX program by declaring them in the function `main` as shown in the following example:

```
main(argc, argv, envp, parm, info)
   int   argc;
   char  *argv[];
   char  *envp[];
   int   parm;
   char  *info;
```

**NOTE** The `envp` parameter is required as a placeholder in the formal parameter list for `main`. This parameter is not initialized on MPE/iX and *must not be used*. It is provided for compatibility with programs on other systems that pass `envp` to `main`.

You invoke the program (called `MYPROG`) with the following command:

```
RUN MYPROG; INFO="STR1 STR2 STR3"; PARM=11
```

The C compiler separates the INFO string into `argv` arguments using blanks as argument delimiters and sets `argc` to the number of `argv` elements. To pass an argument that contains embedded blanks, enclose the argument in quotes. Use single quotes to delimit the argument if the INFO string is enclosed in double quotes; use double quotes if the INFO string is enclosed in single quotes. A quote may be included within a quoted string by escaping the quote with another quote similar to the manner in which the MPE/iX command interpreter allows quotes to be passed in the INFO string. The `argv[0]` argument is set equal to the program name, and `argv[argc]` is set equal to NULL. The PARM and INFO values are passed unchanged to the program. The order of the declaration of parameters to `main` is significant, but the names of the formal parameters can be any valid identifier. For the previous `RUN` command:

```
argv[0] = MYPROG.MYGROUP.MYACCT
argv[1] = "STR1"
argv[2] = "STR2"
argv[3] = "STR3"
argv[4] = NULL
argc    = 4
parm    = 11
info    = "STR1 STR2 STR3"
```

To include blanks within a single entry in the `argv` array, the following command:

```
RUN MYPROG; INFO="STR1 'STR2 WITH BLANKS' STR3"
```

yields:

```
argv[0] = MYPROG.MYGROUP.MYACCT
argv[1] = "STR1"
argv[2] = "STR2 WITH BLANKS"
argv[3] = "STR3"
argv[4] = NULL
argc    = 4
info    = "STR1 'STR2 WITH BLANKS' STR3"
```

To include a single quote in a single quoted argument, the following command:

```
RUN MYPROG; INFO="STR1 'STR2 WITH QUOTE HID' 'DEN' STR3"
```

yields:

```
argv[0] = MYPROG.MYGROUP.MYACCT
argv[1] = "STR1"
argv[2] = "STR2 WITH QUOTE HID'DEN"
argv[3] = "STR3"
argv[4] = NULL
argc    = 4
info    = "STR1 'STR2 WITH QUOTE HID' 'DEN' STR3"
```

A maximum of 1023 `argv[]` entries are allowed and the maximum length of the info string is 279 characters.

## Redirection of Standard Files

The special characters <, >, >>, and & may be specified in the INFO string to redirect standard files for a compiled HP C/iX program. The special characters are described in the following table.

**Table 8-3. Redirection Characters**

| Character | Description |
|---|---|
| < | The name immediately following this symbol in the INFO string is considered a file name and the standard input for the program is read from that file. |
| > | The name immediately following this symbol in the INFO string is considered a file name and the standard output of the program is sent to that file. |
| >> | The name immediately following this symbol in the INFO string is considered a file name and the standard output of the program is **appended** to the end of the file. |
| & | When included after > or >, this character redirects the diagnostic output as well as the standard output to the specified file. The diagnostic output cannot be redirected separately from standard output. |

The redirection characters and the file names which follow these characters do not appear in the `argv` vectors or the `argc` count. For example,

```
RUN MYPROG; INFO= "FILE1 FILE2 >& OUTFILE"
```

runs `MYPROG` with `FILE1` and `FILE2` passed in `argv` and redirects both the diagnostic output and the standard output to the file `OUTFILE.` In this example, `argc` is set to 3.

## The Fileset Wildcard Feature

If the fileset wildcard feature is selected when the program is linked, the INFO string handler for a compiled HP C/iX program expands valid fileset *wildcards* into fully qualified permanent file names and passes them into the main program through the `argv` vectors. To use the wildcard feature, add the relocatable library LIBCWC.LIB.SYS to the FROM list when linking your program. If this file is not in the FROM list, the wildcard feature is not enabled. The CCXLLK and CCXLGO commands do not include this library when linking.

### Example

```
LINK FROM=MYOBJ,LIBCWC.LIB.SYS; RL=LIBCINIT.LIB.SYS; TO=MYPROG
```

This example enables the wildcard feature for INFO strings passed to `MYPROG`.

The fileset wildcards characters for an HP C/iX program are `@`, `?`, and #. They are based on the standard MPE/iX wildcards recognized by commands and subsystems. The wildcard characters are described in the following table.

These characters can be used as follows:

| Character | Description |
|---|---|
| `n@` | Represents all filenames starting with the character `n`. |
| `@n` | Represents all filenames ending with the character `n`. |
| `n@x` | Represents all filenames starting with the character `n` and ending with the character `x`. |
| `n` | Represents all filenames starting with the character `n` followed by a maximum of seven digits. |
| `?n@` | Represents all filenames whose second character is `n`. |
| `n?` | Represents all 2-character filenames starting with the character `n`. |
| `?n` | Represents all 2-character filenames ending with the character `n` |

**NOTE** Each wildcard character is one of the 8-character limit for *account*, *group*, and *filename*. A valid wildcard fileset must start with an alphabetic character, an `@`, or a `?`. Invalid wildcard filesets are passed unaltered to the main program.

### Example

If the permanent files in the group and account `MYGROUP.MYACCT` contains `FILE1`, `FILEX`, `MYFILE`, and `MYPROG`, the following:

```
RUN MYPROG; INFO="@ FILE# FILE?"
```

yields:

```
argv[0] = "MYPROG.MYGROUP.MYACCT"    /* name of program */
argv[1] = "FILE1.MYGROUP.MYACCT"     /*     @ */
argv[2] = "FILEX.MYGROUP.MYACCT"     /*     @ */
argv[3] = "MYFILE.MYGROUP.MYACCT"    /*     @ */
argv[4] = "MYPROG.MYGROUP.MYACCT"    /*     @ */
argv[5] = "FILE1.MYGROUP.MYACCT"     /* FILE# */
argv[6] = "FILE1.MYGROUP.MYACCT"     /* FILE? */
argv[7] = "FILEX.MYGROUP.MYACCT"     /* FILE? */
argv[8] = NULL
argc    = 8
info    = "@ FILE# FILE?"
```

If no files are found in the fileset, or an error occurs in attempting to expand a wildcard fileset, a diagnostic message is printed to $STDLIST and the process terminates immediately. Attempting to use a wildcard fileset that would expand the number of argv elements beyond the 1023 element maximum also results in an error.

## Escaping Special INFO String Characters

Special INFO string characters are characters that have special meaning to the C INFO string parser. They include the standard I/O redirection characters <, >, and &, as well as the fileset wildcard characters @, ?, and # if the wildcard feature is enabled. If you wish to pass strings containing these characters into a program in the argv vectors, you must enclose these strings with either single or double quotes. Use single quotes if your INFO string is delimited with double quotes, and double quotes if your INFO string is delimited with single quotes. This will disable their special functions and cause the INFO string parser to pass them along to the main program.

### Example

```
RUN MYPROG; INFO ="'@' '<IDENTIFIER>'"
```

yields:

```
argv[0] = "MYPROG.MYGROUP.MYACCT"
argv[1] = "@"
argv[2] = "<IDENTIFIER>"
argv[3] = NULL
argc    = 3
info    = "'@' '<IDENTIFIER>'"
```

## HP C/iX and Job Control Words

When an HP C/iX program terminates normally, a special job control word, CJCW, is set to the exit value of the program. It contains the value of the parameter passed to the C library routine exit, or _exit, or the value returned from the function main if exit is not explicitly called. The convention for C programs is to exit with a nonzero value if an error condition has occurred, or exit with zero if no errors have occurred. CJCW could be used for checking the exit value of a program in a job file, or programmatically checking the exit value of a child process. The value of CJCW is unpredictable if the function main does not take care to return a value or if the exit function is not explicitly called. **Examples**

```
main() {
   .
   .
   exit(7);
}
```

or

```
main() {
   .
   .
   return(7);
}
```

Either of the above examples sets CJCW to the value of 7 on program termination.

If a C program calls the C library routine `abort`, the system job control word JCW is set to FATAL and a diagnostic message is printed to `$STDLIST`. CJCW is set to a nonzero value.

## Arithmetic Traps

C/iX programs execute with all arithmetic traps disabled. The C program startup routines call the `ARITRAP` intrinsic with an argument of zero to disable the traps.

## Special Preprocessor Considerations

The HP C/iX compiler first executes a preprocessing phase during which all preprocessor directives (lines beginning with #) are interpreted and acted upon. See chapter 7 for detailed information about preprocessing. This section describes features unique to the preprocessing phase on MPE/iX systems.

### Preprocessor Output File

During the preprocessing phase, an output file is created for use by the compiler. Although the file is created with a record size and file limit that is sufficiently large for most applications, it is possible for a source program to expand beyond the default file limit and record size. You can use two HP C/iX compiler options, `+F` and `+C`, to overcome this problem.

Typically, the file limit may be exceeded if you compile a very large source file or if you use many `#include` directives. The compiler issues an error message indicating that the file limit of the preprocessor output file was exceeded. To resolve this error, you must recompile the source file using the `+F` compiler option to specify a larger file limit. Any unused file space is returned when the output file is closed so you should specify a sufficiently large file limit.

A source file might have a line that expands beyond the default record size if it contains a large macro, a number of macros, macros that include other macros, or any combination of these conditions. This problem is more difficult to detect because many lines can exceed the default size without causing an error. It is only when a token, such as an identifier, is split across two records in the output file that an error occurs. The error occurs on the line that contains the expanded macro. To resolve this type of error, recompile using the `+C` option to specify a larger record size, or reduce the size of the macro. The default record size is 512 bytes.

### Predefined Macros

The preprocessor on MPE/iX has one predefined macro, `mpexl`, to aid the identification and isolation of system dependent code. This macro behaves as if the preprocessor directive `#define mpexl 1` is included at the top of the file. The `mpexl` macro is typically used with the `#ifdef` statement.

#### Example

```
#ifdef mpexl
   MPE/iX specific code ...
#endif
```

# 9 HP C/iX Implementation Topics

This chapter describes topics that are specific to programming in C on MPE/iX.

The following topics are included:

- Data types
- Bit-fields
- IEEE floating-point format
- Lexical elements
- Qualifying structures and union references
- Data alignment
- Type mismatches in external names
- Expressions
- Pointers
- Array limits
- Scope of extern declarations
- Conversions among real numbers
- Variable length argument lists
- Include file locations

## Data Types

Data types are implemented in HP C/iX as follows:

- The `char` type is signed.
- All types can have the `register` storage class, although it is only honored for scalar types. Ten register declarations per function are honored. More are honored when the +R option is used.
- The signed integer types are represented internally using twos complement form.
- Structures (and unions) start and end on the alignment boundary of their most restrictive member.

Table 9-1 lists the sizes and ranges of different HP C/iX data types.

Refer to the *HP C Programmer's Guide* for comparisons of data storage and alignment on the following computer systems:

- HP 3000 Series 900
- HP 3000/V
- HP 9000 Series 300/400
- HP 9000 Series 700/800

**Table 9-1. HP C/iX Data Types**

| Type | Bits | Bytes | Low Bound | High Bound | Comments |
|---|---|---|---|---|---|
| `char` | 8 | 1 | -128 | 127 | Character |
| `signed char` | 8 | 1 | -128 | 127 | Signed integer |
| `unsigned char` | 8 | 1 | 0 | 255 | Unsigned integer |
| `short` | 16 | 2 | -32,768 | 32,767 | Signed integer |
| `unsigned short` | 16 | 2 | 0 | 65,535 | Unsigned integer |
| `int` | 32 | 4 | -2,147,483,648 | 2,147,483,647 | Signed integer |
| `unsigned int` | 32 | 4 | 0 | 4,294,967,295 | Unsigned integer |
| `long` | 32 | 4 | -2,147,483,648 | 2,147,483,647 | Signed integer |
| `unsigned long` | 32 | 4 | 0 | 4,294,967,295 | Unsigned integer |
| `float` | 32 | 4 | See (a) below. | See (b) below. | Floating-point |
| `double` | 64 | 8 | See (c) below. | See (d) below. | Floating-point |
| `long double` | 128 | 16 | See (e) below. | See (f) below. | Floating-point |
| `enum` | 32 | 4 | -2,147,483,648 | 2,147,483,647 | Signed integer |

### Comments

In the following comments, the low bounds of `float`, `double`, and `long double` data types are given in their **normalized** and **denormalized** forms. Normalized and denormalized refer to the way data is stored. Normalized numbers are represented with a greater degree of accuracy than denormalized numbers. Denormalized numbers are very small numbers represented with fewer significant bits than normalized numbers.

a. Least normalized: 1.17549435E-38F
   Least denormalized: 1.4012985E-45F

b. 3.40282347E+38F

c. Least normalized: 2.2250738585072014E-308
   Least denormalized: 4.9406564584124654E-324

d. 1.7976931348623157E+308

e. Least normalized: 3.3621031431120935062626778173217526026E-4932L
   Least denormalized: 6.4751751194380251109244389582276465525E-4966L

f. 1.1897314953572317650857593266280070162E+4932L

## Bit-Fields

- Bit-fields in structures are packed from left to right (high-order to low-order).
- The high order bit position of a "plain" integer bit-field is treated as a sign bit.
- Bit-fields of types `char`, `short`, `long`, and `enum` are allowed.
- The maximum size of a bit-field is 32 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:

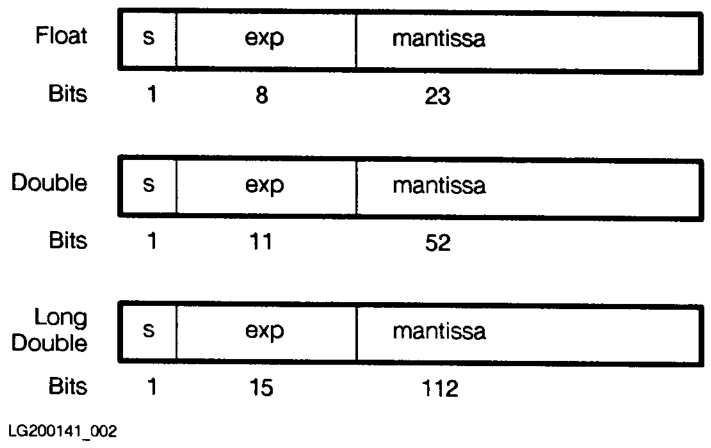
  -2,147,483,648 to 2,147,483,647 for 32-bit signed quantities

  0 to 4,294,967,295 for 32-bit unsigned quantities
- 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 9-1.

**Figure 9-1. 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 9-2. HP C/iX Data Types**

| | float | double | long double |
|---|---|---|---|
| **bias** | +127 | +1023 | +16383 |
| $\mathbf{E}_{max}$ | +127 | +1023 | +16383 |
| $\mathbf{E}_{min}$ | -126 | -1022 | -16382 |

$E_{min}$-1 is used to encode 0 and denormalized numbers.

$E_{max}$+1 is used to encode infinities and NaNs.

NaNs are binary floating-point numbers that have all ones in the exponent and a nonzero fraction. NaN is the term used for a binary floating-point number that has no value (that is, "Not A Number").

If `E` is within the range

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

the mantissa field contains the number in a normalized form, preceded by an implicit 1 and binary point.

In accordance with the IEEE standard, floating-point operations are performed with traps not enabled, and the result of such an operation is that defined by the standard. This means, for example, that dividing a positive finite number by zero will yield positive infinity, and no trap will occur. Dividing zero by zero or infinity by infinity will yield a NaN, again with no trap. For a discussion of infinity arithmetic and operations with NaNs, in the context of the IEEE standard, see the *HP Precision Architecture and Instruction Set Reference Manual*.

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.

## 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.

## Data Alignment Pragma

This section discusses the `HP_ALIGN` data alignment pragma and the differences between the data alignment architectures of the MPE/iX and MPE/V systems.

The `HP_ALIGN` data alignment pragma allows you to control the alignment of fields within structures and unions. It facilitates the interchange of data between MPE systems having different data alignment architectures.

Data alignment architectures specify the number of bits allocated to store various data types, whether a data type is aligned on a byte, two-byte, word, or double word boundary, and padding among bit-fields. The differences in data alignment architectures are especially important to consider when passing data from one machine to another.

The `HP_ALIGN` pragma facilitates transferring data among MPE V and MPE/iX systems and when accessing TurboImage databases from HP C/iX.

The syntax for the `HP_ALIGN` pragma is:

[#pragma HP_ALIGN {MPE_16POP}]

The `MPE_16` option directs the HP C/iX compiler to set the alignment of `int`, `float`, and `double` in structures and unions to be aligned according to the MPE/V alignment scheme. This option also sets the alignment of structures and unions to start and end on at least a half-word boundary.

The `POP` option turns off the `HP_ALIGN` pragma and alignment reverts to word (32-bit) alignment. For example:

```
#pragma HP_ALIGN MPE_16
struct {char a; double b; int c;} d;
#pragma HP_ALIGN POP
```

### Comparison of MPE/V and MPE/iX Data Alignment

MPE/V systems align data within structures and unions differently than MPE/iX systems. The data alignment rules for these two systems are described using the following code fragment for comparison purposes:

**Figure 9-2. Code Fragment for Comparing Storage and Alignment**

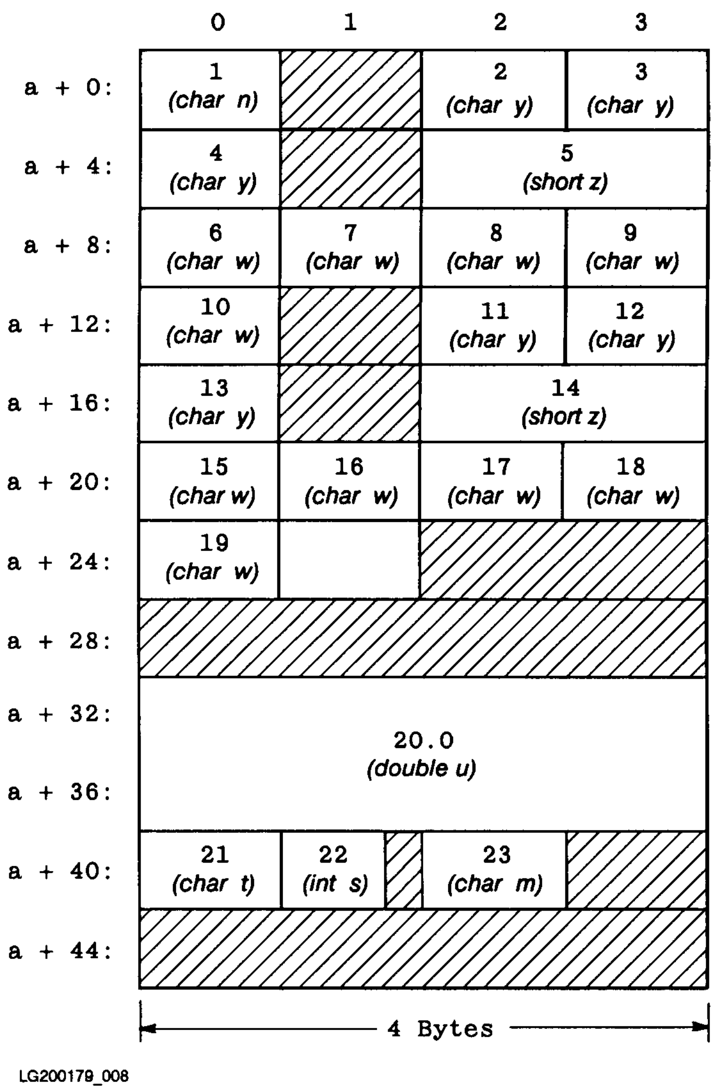
```
struct x {
   char y[3];
   short z;
   char w[5];
};

struct q {
   char n;
   struct x v[2];
   double u;
   char t;
   int s:6;
   char m;
} a = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,
       20.0,21,22,23};
```

### Native Data Alignment on HP C/iX

Figure 9-3 shows how the data in Figure 9-2. on page 154 is stored in memory when using HP C/iX. The values are shown above the variable names. Memory locations containing shading are padding bytes.

**Figure 9-3. Storage with HP C/iX**



The structure `a` is aligned on an 8-byte boundary because the most restrictive data type within the structure is the `double u`.

Table on page 153 shows the padding for the example code fragment:

**Table 9-3. Padding on HP 9000 Series 700/800 and HP 3000 Series 900**

| Padding Location | Reason for Padding |
|---|---|
| a+1 | Aligns the structure `x` on a 2-byte boundary because the most restrictive member is `short`. |

**Table 9-3. Padding on HP 9000 Series 700/800 and HP 3000 Series 900**

| Padding Location | Reason for Padding |
|---|---|
| a+5 | Aligns the short z on a 2-byte boundary. |
| a+13 | Fills out the struct x to a 2-byte boundary. |
| a+17 | Aligns the short z on a 2-byte boundary. |
| a+25 | Fills out the structure to a 2-byte boundary. |
| a+26 through a+31 | Aligns the double u on an 8-byte boundary. The bit-field s begins immediately after the previous item at a+41. Two bits of padding is necessary to align the next byte properly. |
| a+43 through a+47 | Fills out the struct q to an 8-byte boundary. |

### Data Alignment Using HP_ALIGN MPE_16

When the sample code fragment is compiled and run using the HP_ALIGN MPE_16 pragma, data is stored as shown in Figure 9-4. on page 157.

**Figure 9-4. Storage Using the HP_ALIGN MPE_16 Pragma**

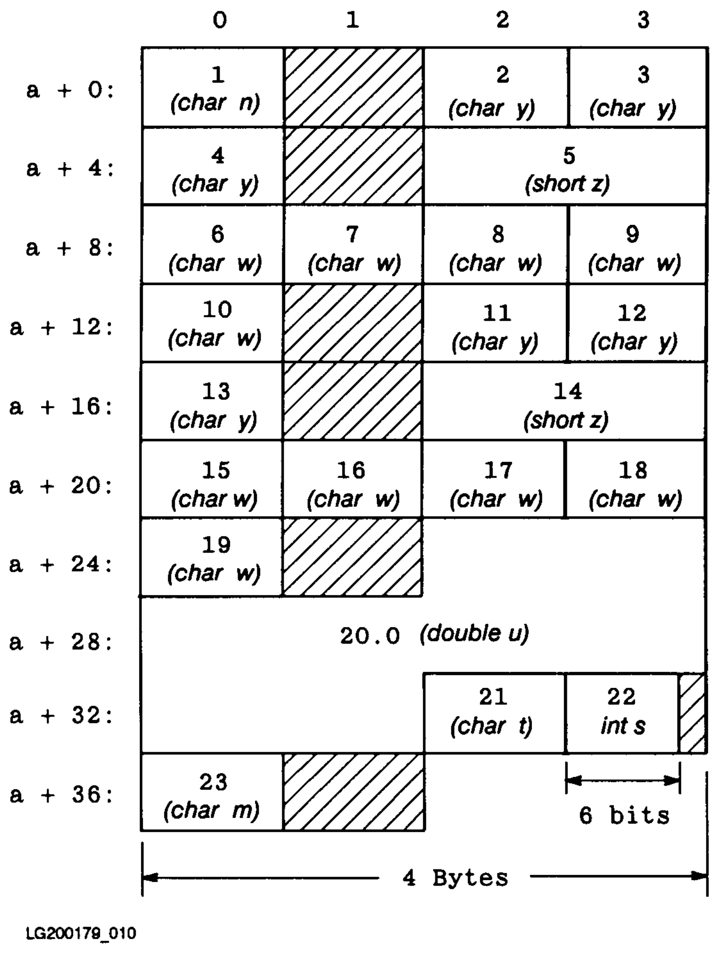


Table on page 157 shows the padding for the example code fragment when using `HP_ALIGN MPE_16`:

**Table 9-4. Padding Using the HP_ALIGN MPE_16 Pragma**

| Padding Location | Reason For Padding |
|---|---|
| `a+1` | Within structures, align structure `x` on a 2-byte boundary. |
| `a+5` | Aligns the `short z` on a 2-byte boundary. |
| `a+13` | Structures within structures are aligned on a 2-byte boundary. |
| `a+17` | Aligns the `short z` on a 2-byte boundary. |
| `a+25` | Doubles are 2-byte aligned within structures. |
| `a+36` | Aligns `char m` on a byte boundary. |

The differences between the `HP_ALIGN MPE_16` and the native MPE/iX alignments are:

- MPE/V aligned records are aligned on a 2-byte boundary. MPE/iX aligned records are aligned according to the most restrictive data type within the structure.
- MPE/V aligned doubles are 2-byte aligned. MPE/iX aligned doubles are 8-byte aligned within structures.
- MPE/V aligned long doubles, available in ANSI mode only, are 2-byte aligned. MPE/iX aligned long doubles are 8-byte aligned within structures.
- MPE/V aligned enumerated data types are 2-byte aligned in a structure, array, or union. MPE/iX aligned enumerated types are always 4-byte aligned.

## Pointers to Half-Word Aligned Data Items

Pointers in HP C for the HP 3000 Series 900 by default point to objects aligned on 32-bit word addresses. By default, the machine instructions for pointers generated by the compiler depend on the data being word aligned. A run-time memory fault occurs if a pointer to a word-aligned data item accesses a half-word (16-bit) aligned data item. Using non-natively aligned structures can produce this situation.

In the following example, a run-time memory fault occurs under normal conditions because `pi` points to a half-word aligned field.

```
#pragma HP_ALIGN MPE_16
struct { char a; int b; } c;
int *pi;

pi = c.b; /* Now pi points to a half-word aligned integer. */
```

The `+u` compiler option resolves this by causing the compiler to generate code to access pointers with half-word addressing.

For best performance you should encapsulate this option, and after reading a data item into a half-word aligned structure, immediately copy the contents, member by member to a word aligned structure.

## Long and Short Pointers

HP C defines two classes of data pointers: short and long pointers. A *short pointer* is a 32-bit pointer that contains the offset of an object local to its process. A *long pointer* is a 64-bit pointer that may point to an object outside its current process space. The high-order 32 bits of a long pointer contain a space ID number, and the low-order 32 bits represent an offset within the space defined by that space ID.

A short pointer can point to any addressable object within its own process space. This includes local and global variables, function parameters, and heap variables. Long pointers may point to any addressable object on the system. This includes objects that are outside the space of the current process. Long pointers are useful for calling system intrinsics that require long pointer parameters (such as PRINT), and for accessing user-mapped files.

### Miscellaneous Pointer Features

- 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.
- `fp()` is equivalent to  in an expression when `fp` is of type pointer to function.

### Declaring Long Pointers

You declare long pointer variables or parameters using the normal (short pointer) syntax *except* you need to substitute a caret (^) for the asterisk(*). Refer to chapter 3 for information on declaring pointers.

For example:

```
/* int_pointer is a long pointer to an integer */
int ^int_pointer;

/* char_pointer is a long pointer to a character */
char ^char_pointer;
```

Long pointers may not be declared with initializers. Therefore, a statement such as:

```
       char ^long_ptr = "some string";
```

is not allowed. Also, long function pointers are not allowed.

### Using Long Pointers

Long pointers are dereferenced the same as short pointers. If `p` is declared as a long pointer to an `int`, such as `int ^p;`, then `*p` returns the contents of that integer. If `p` is declared as a pointer to a structure containing a member `mem`, `p->mem` will return the contents of that member.

To produce a long pointer that points to the same object as a short pointer, you can use an assignment statement. For example, after the assignment statement in the following C code, `long_ptr` and `short_ptr` both point to the same object.

```
int ^long_ptr;  /* long_ptr is a long pointer to an integer */
int *short_ptr;  /* short_ptr is a short pointer to an integer */
    .
    .
    .
long_ptr = short_ptr;
```

After the assignment statement, the low-order 32 bits of the long pointer are identical to the original short pointer.

A short pointer may not be assigned the contents of a long pointer unless the long pointer points to an object within the space of the current process.

You can apply the unary operator & to a long pointer to get the address of the pointer. The resulting type of the operation &`long_ptr` is a short pointer. The following example assigns the address of a long pointer to a variable declared as a short pointer to a long pointer.

```
int ^long_ptr;
int ^*sprt_to_lptr;

sptr_to_lptr = &long_ptr;
```

Note that the variable `sptr_to_lptr` must be declared as a short pointer to a long pointer because the type of &`long_ptr` is a short pointer.

Standard pointer arithmetic and pointer subscripting may be applied to long pointers, but only the low-order 32 bits are used and modified.

Comparison of long pointers using the relational operators <, <=, >, and >= compares only the low order 32 bits, the offset portion. The high order 32 bits, the ID portion, are ignored. Comparison of long pointers using the relational operators, == and !=, compares both the low order 32 bits and the high order 32 bits.

Assignment of zero (NULL) to a long pointer assigns zero to the high order 32 bits and to the low order 32 bits.

Casts can be applied to long pointers but you must observe the standard alignment restrictions. For example, a long pointer to a character should not be cast to a long pointer to an integer because an integer has more restrictive alignment requirements than a character. Integers are aligned on 4 byte boundaries and characters are aligned on byte boundaries.

There are no standard conversion rules for passing long pointer parameters to other routines. Therefore, no implicit conversions are performed when these parameters are passed. That is, short pointers are not converted to long pointers when a function call is made, and vice versa.

All pointers passed to standard C library routines, including the heap manager routines, must be short pointers.

## Expressions

The value of an expression that overflows or underflows is undefined, except when the operands are unsigned.

## 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 of 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.
- HP C/iX supports the `#line` *`digit-sequence`* `"filename"` directive. This directive is used to set the line number and file name for compile time diagnostics.
- See chapter 7 "Preprocessing Directives."

## 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. See the *HP C/iX Library Reference Manual* for more details on <stdarg.h>. The <varargs.h> header is retained for compatibility with pre-ANSI compilers and earlier releases of HP C/iX.

To use varargs, a program must include the header <varargs.h>. A function that expects a variable number of arguments must declare the first variable argument as va_alist in the function declaration. The macro va_dcl must be used in the parameter declaration.

A local variable should be declared of type va_list. This variable is used to point to the next argument in the variable argument list.

The va_start macro is used to initialize the argument pointer to the initial variable argument.

Each variable argument is accessed by calling the va_arg macro. This macro returns the value of the next argument, assuming it is of the specified type, and updates the argument pointer to point to the next argument.

The va_end macro is provided for consistency with other implementations; it performs no function on the 900 Series HP3000 computers.

The following example demonstrates the use of the <varargs.h> header:

### Example

```
#include <varargs.h>
#include <stdio.h>

enum arglisttype {NO_VAR_LIST, VAR_LIST_PRESENT};
enum argtype     {END_OF_LIST, CHAR, DOUB, INT, PINT};

int foo (va_alist)
va_dcl   /* Note: no semicolon */
{
    va_list ap;
    int a1;
    enum arglisttype a2;

    enum argtype ptype;
    int i, *p;
    char c;
    double d;

    /* Initialize the varargs mechanism */
    va_start(ap);

    /* Get the first argument, and arg list flag */
    a1 = va_arg (ap, int);
    a2 = va_arg (ap, enum arglisttype);

    printf ("arg count = %d\n", a1);

    if (a2 == VAR_LIST_PRESENT) {
 /* pick up all the arguments */
 do {
     /* get the type of the argument */
     ptype = va_arg (ap, enum argtype);

     /* retrieve the argument based on the type */
     switch (ptype) {
  case CHAR:  c = va_arg (ap, char);
       printf ("char = %c\n", c);
       break;

  case DOUB:  d = va_arg (ap, double);
       printf ("double = %f\n", d);
       break;

  case PINT:  p = va_arg (ap, int *);
       printf ("pointer = %x\n", p);
       break;

  case INT :  i = va_arg (ap, int);
       printf ("int = %d\n", i);
       break;

  case END_OF_LIST :
       break;

  default:    printf ("bad argument type %d\n", ptype);
       ptype = END_OF_LIST; /* to break loop */
       break;
     } /* switch */
 } while (ptype != END_OF_LIST);
    }

    /* Clean up */
    va_end (ap);
}

main()
{
    int x = 99;

    foo (1, NO_VAR_LIST);
    foo (2, VAR_LIST_PRESENT, DOUB, 3.0, PINT, &x, END_OF_LIST);
}
```

## Location of Files

The following table lists the location of files used in compiling, linking, and running HP C/iX programs.

**Table 9-5. Location of Files**

| File or Library | Location |
|---|---|
| CCXL | `CCXL.PUB.SYS` |
| CCXLLK | `CCXLLK.PUB.SYS` |
| CCXLGO | `CCXLGO.PUB.SYS` |
| CCSTDRL | `CCSTDRL.LIB.SYS` |
| Compiler | `CCOMXL.PUB.SYS` |
| Preprocessor (non-ANSI) | `CPP.PUB.SYS` |
| Preprocessor (ANSI) | `CPPANSI.PUB.SYS` |
| Error catalog | `CCMSGCAT.PUB.SYS` |
| Linker | `LINKEDIT.PUB.SYS` |
| C Library (RL) | `LIBC.PUB.SYS` |
| C Library (XL) | `XL.PUB.SYS` |
| C Library (XL initializer) | `LIBCINIT.LIB.SYS` |
| Wildcard feature | `LIBCWC.LIB.SYS` |
| LIBCRAND | `LIBCRAND.LIB.SYS` |
| XDB end info | `XDBEND.LIB.SYS` |
| Math library | `LIBM.LIB.SYS` |
| Math library (ANSI version) | `LIBMANSI.LIB.SYS` |
| Additional ANSI library | `LIBCANSI.LIB.SYS` |

# 10 Using Intrinsics

This chapter describes the use of intrinsic functions in HP C/iX programs.

System routines on the MPE/iX operating system are generally referred to as *intrinsics* because they are an integral or "intrinsic" part of the operating system. The essential characteristic of an intrinsic is that a description of its interface is stored, in a compiled form, in a specially formatted file known as an *intrinsic file*. MPE/iX intrinsics are described in the file `SYSINTR.PUB.SYS.` Additionally, you can define your own intrinsics and store their descriptions in your own intrinsic files, using the HP Pascal/iX compiler.

You use the `intrinsic` pragma to declare that the calling conventions for a particular function are to be found by the compiler in an intrinsic file. You use the `intrinsic_file` pragma to give the name of the intrinsic file, if it is other than `SYSINTR.PUB.SYS`.

An advantage of declaring a system routine as an intrinsic is that it often simplifies the function call in the HP C source program. For example, parameters may be optional; they can be omitted from the call and the compiler will generate the appropriate default values. Furthermore, any "hidden" parameters required by the intrinsic will be generated automatically. Finally, the compiler checks the types of the actual arguments against the types of the intrinsic parameters and issues diagnostics if mismatches are found.

## Intrinsic Pragma

You use the `intrinsic` pragma to declare an external function as an intrinsic. It has the following format:

```
#pragma intrinsic intrinsic-name1 [user-name] [,intrinsic-name2 [user-name]
]...
```

Where:

| | |
|---|---|
| *intrinsic-name* | is the name of the intrinsic you want to call. |
| *user-name* | is any valid C identifier. If specified, you must use this name to invoke the intrinsic from the source program. |

### Examples

```
#pragma intrinsic FOPEN
#pragma intrinsic FCLOSE myfclose
#pragma intrinsic FCHECK, FGETINFO
#pragma intrinsic FWRITE mpe_fwrite, FREAD mpe_fread
```

The first example shows how to declare the FOPEN intrinsic as an external function. The second example shows how to declare FCLOSE; you must call it by the name `myfclose` in your program. The third and fourth examples each declare two intrinsics. The fourth provides alternative names for the intrinsics.

When you designate an external function as an intrinsic, the compiler refers to the intrinsic file to determine the function type, the number of parameters, and the type of each parameter. The compiler then uses this information to perform the necessary conversions and insertions to correctly invoke the routine, or to issue warnings and errors if proper invocation is not possible.

Specifically, for intrinsic calls, the HP C/iX compiler does the following:

- Converts all value parameters to the type expected by the intrinsic function. Conversions are performed as if an assignment is done from the argument value to the formal parameter. This is known as `assignment conversion`. If a value cannot be converted, an error message is issued.
- Converts addresses passed as reference parameters to the proper address type. This means that short addresses are coerced to long addresses as required by the intrinsic function. An integer value of zero is considered a legal value (NULL) for any address parameter.
- Allows missing arguments in the call to the intrinsic if the intrinsic defines default values for those parameters. The compiler supplies the default values for the missing arguments, or issues an error message if there is no defined default value. Missing arguments are allowed within an argument list or at the end of an argument list.
- Issues an error message if there are too many arguments.

- Inserts "hidden" arguments required to correctly call Pascal routines that have ANYVAR parameters (size is hidden) or that are EXTENSIBLE (parameter count is hidden).

If you declare a system intrinsic using an `extern` declaration rather than an intrinsic program, and if you do not provide a function prototype, none of the above checks, conversions, or insertions are done. The address of an intrinsic can be taken, but if a call is made using a pointer to the intrinsic, the above checks are not performed. The intrinsic call then degenerates into a normal C function call.

To ensure that all calls are handled correctly by the compiler, the intrinsic pragma should declare the name of the intrinsic using an identifier with the identical case that is used by the function calls in the program, especially if the optional user-name is not specified. No other functions in the program should have the same name as any intrinsic that is declared, regardless of the case. This is because the actual run time symbol used to call an intrinsic is not necessarily the same case as the identifier used to declare that intrinsic.

There are a few HP C library routines, such as `fopen` or `fwrite`, that have the same names as intrinsic functions. If you wish to call any of these C library routines that always use lower case identifiers from a program that also calls the intrinsics with the same name, the intrinsics must be declared and called using mixed case or upper case identifiers, or by using the optional user-name.

## Intrinsic_file Pragma

The `intrinsic_file` pragma specifies the name of the file in which the compiler can locate information about intrinsic functions. It has the following format:

```
#pragma intrinsic_file "filename"
```

where *`filename`* is the fully qualified filename of the file you want the compiler to use to look up information about intrinsics declared using the `intrinsic` pragma. If you do not include the group and account name of the file, the compiler looks in the group and account of the current user.

If you do not use this pragma, the compiler looks in a file called SYSINTR.PUB.SYS by default. The file SYSINTR.PUB.SYS contains descriptions of all MPE/iX intrinsics. You only need to use the `intrinsic_file` pragma if you are building your own intrinsic files using the HP Pascal/iX compiler and you must specify a file other than the default. Refer to the *HP Pascal Programmer's Guide* for information about building your own intrinsic files.

The compiler refers to the specified file until another `intrinsic_file` pragma is encountered. To return the search to SYSINTR.PUB.SYS, specify the `intrinsic_file` pragma with a null string:

```
#pragma intrinsic_file
```

Some examples of `intrinsic_file` and `intrinsic` pragmas follow.

```
#pragma intrinsic FOPEN, FCLOSE, FREAD /* SYSINTR.PUB.SYS used */
#pragma intrinsic_file "MYINTR"
#pragma intrinsic mytest1, mytest2 /* MYINTR.MYGROUP.MYACCT used */
#pragma intrinsic_file ""
#pragma intrinsic FCHECK, FGETINFO /* SYSINTR.PUB.SYS used */
```

In the first pragma, the compiler searches the default file for information about the `FOPEN`, `FCLOSE`, and `FREAD` pragmas. The second pragma specifies a different file for the compiler to search, `MYINTR`. The third pragma declares two intrinsics, `mytest1` and `mytest2`, that must be described in `MYINTR`. The fourth pragma returns the search to `SYSINTR`, where `FCHECK` and `FGETINFO` descriptions are found.

## Condition Codes

Condition codes are temporary values that provide basic information about the execution of intrinsics. Many of the MPE/iX intrinsics alter the condition code upon their completion. You can review condition code values to determine the success of an intrinsic call. To recover the condition code, call the HP C/iX library routine, `ccode`, immediately upon returning from an intrinsic. This ensures that no other instruction alters the condition code. You should only use the `ccode` routine in simple `if` statements or assignment statements because of the possible side effects of the order of expression evaluation.

The following macros in the MPE.H header file define condition code values:

**Table 10-1. Condition Code Values**

| Macro | Value | Meaning |
|---|---|---|
| `CCG` | Condition code greater | A special condition occurred but may not have affected the execution of the request. |
| `CCL` | Condition code less | The request was not granted, but the error condition may be recoverable. |
| `CCE` | Condition code equal | This generally indicates that the request was granted. |

The specific meaning of the condition code depends on which intrinsic function is called. Refer to the description of each intrinsic in the *MPE/iX Intrinsics Reference Manual* for the exact meaning of the condition code.

### Example

The following code segment illustrates typical condition code checking.

```
SOME_INTRINSIC_FUNC(arg1, arg2);
if (ccode() != CCE)
   error_handler();  /* Intrinsic call failed; */
                     /* set up error handler   */
```

## Calling Trap Intrinsics

The MPE/iX trap intrinsics often require a *plabel* parameter that is either by value or by reference. The HP C/iX implementation of plabel is simply the "pointer to function" type. Whenever a function pointer is used, the compiler is actually generating a plabel. Since plabel is not a basic data type across languages, and other languages do not support the integer value zero as a valid null pointer, the intrinsic mechanism describes plabels as integers in both the documentation and SYSINTR file. This means that a cast must be applied to the plabel parameters for trap intrinsics that are called from C, or the compiler issues an error. Remember that using a function name in a context other than a function call yields "pointer to function."

### Example

```
#pragma intrinsic XCONTRAP
#include <stdio.h>
void func1(void)
{
   printf("control-Y was hit\n");
}

main(void)
{
void (*oldfunc1)();

   /* First parm is plabel by value, */
   /* second plabel is by reference. */
   XCONTRAP( (int)func1, (int*)&oldfunc1 );
}
```

# 11 The Listing Facility

The HP C/XL compiler generates a listing by default. This listing appears on the $STDLIST file, or it can be redirected to a file as explained in chapter 8.

## Listing Format

The listing consists of the following information:

- A banner on the top of each page.
- A line number for each source line.
- The nesting level for each statement or declaration.

There are two styles of listing available: *non-ANSI mode* and *ANSI mode*.

### Non-ANSI Mode

In non-ANSI mode, the text of the listing is the output of the preprocessor after macro substitution with `#include` files inserted.

### ANSI Mode

In ANSI mode, the text of the listing is the original version of the source file before macro substitution; `#include` files are inserted. To produce the non-ANSI style listing, compile with the `+Lp` option instead of the `+L` option.

**NOTE** The `+Lp` option only has this effect when it is used in conjunction with the `-Aa` option.

In either mode, comments are stripped from the listing (unless the `-C` option is specified).

## Listing Pragmas

The listing facility provides a number of pragmas to control various aspects of the listing format. The available pragmas are described below.

`#pragma LINES` *`linenum`*

Sets the number of lines per page to *`linenum`*. Default is 63. Minimum number is 20 lines.

`#pragma WIDTH` *`pagewidth`*

Sets the width of the page to *`pagewidth`*. Default is 80 columns. Minimum number is 50 columns.

**NOTE** If the WIDTH pragma is being used, put it before any TITLE or SUBTITLE pragmas, since the title and subtitle field widths vary with the page width.

`#pragma TITLE` *`"string"`*

Sets the page title to *`string`*. *`string`* is truncated without warning to 44 characters less than the page width. Default is the empty string.

`#pragma SUBTITLE` *`"string"`*

Sets the page subtitle to *`string`*. *`string`* is truncated without warning to 44 characters less than the page width. Default is the empty string.

**NOTE** The TITLE and SUBTITLE pragmas do not take effect until the second page, because the banner on the first page appears before the pragmas.

`#pragma PAGE`

Causes a page break and the start of a new page.

[#pragma AUTOPAGE {ONOFF}]

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

[#pragma LIST {ONOFF}]

Turns the listing `ON/OFF`. The default is `ON`. Use this pragma as a toggle to turn listing off around any source lines that you do not want to be listed, such as include files.

## Listing Options

Two compiler options are provided to write additional information to the listing. The `-Wc,-m` (abbreviated `+m`) option is used to generate identifier maps. The `-Wc,-o` (abbreviated `+o`) option is used to generate code offsets.

### Identifier Maps

When the `+m` option is specified, the compiler produces a series of identifier maps, grouped by function. The map shows the declared identifiers, storage class, type, and address or constant value.

The first column of the map lists, in alphabetical order, the initial 20 characters of all the identifiers declared in the function. Member names of structures and unions appear indented under the structure or union name.

The second column displays the storage class of each identifier. The compiler distinguishes the following storage classes:

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

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

```
array          int                 union
char           long int            unsigned char
double         long double         unsigned int
enum           short int           unsigned long
float          signed char         unsigned short
function       struct              void
```

The type qualifiers, `const` and `vol` (for volatile), can also appear in the third column.

The fourth column indicates the relative register location of an identifier. Members of a union type are in the form `W @ B`, where `W` is the byte offset and `B` is the bit offset within the word. Both offsets are given in hexadecimal notation. **Example**

```
main(void)
{
   enum colors {red, green, blue} this_color;
   struct SS {
      char           *name;
      char           sex;
      int            birthdate;
      int            ssn;
      float          gpa;
      struct SS      *previous;
      } pupil_rec;

   union UU {
      int        flag;
      float      average;
      } datum;
   struct SS second_pupil;

this_color = red;
pupil_rec.sex = 'm';
datum.flag = 1;
second_pupil.gpa = 3.72;

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

Identifier         Class       Type               Address
         -                     -
 main              ext def     int ()             main
         L O C A L   I D E N T I F I E R   M A P S


                             main

    Identifier    Class       Type               Address
        -                     -
    blue          const       enum colors        2
    datum         auto        union UU           SP-64
      flag        member      int                  0x0 @ 0x0
      average     member      float                0x0 @ 0x0
    green         const       enum colors        1
    pupil_rec     auto        struct SS          SP-60
      name        member      char   *             0x0 @ 0x0
      sex         member      char                 0x4 @ 0x0
      birthdate   member      int                  0x8 @ 0x0
      ssn         member      int                  0xc @ 0x0
      gpa         member      float                0x10 @ 0x0
      previous    member      struct  *            0x14 @ 0x0
    red           const       enum colors        0
    second_pupil  auto        struct SS          SP-88
      name        member      char   *             0x0 @ 0x0
      sex         member      char                 0x4 @ 0x0
      birthdate   member      int                  0x8 @ 0x0
      ssn         member      int                  0xc @ 0x0
      gpa         member      float                0x10 @ 0x0
      previous    member      struct  *            0x14 @ 0x0
    this_color    auto        enum colors        SP-36
```

## Code Offsets

When the `-Wc,-o` (or `+o`) option is specified, the compiler produces a series of the code offsets for each executable statement, grouped by function. Source line numbers are given in decimal notation followed by the associated code address specified in hexadecimal notation. The code address is relative to the beginning of the function.

### Example

```
#include <stdio.h>
main(void)
{
   int   j;
   void  func1 (int);
   void  func2 (int);

   for (j=0;  ) {
      func1 (j);
      func2 (j);
   }
}

void func1 (int i)
{
   while (i ) {
      if (!(i % 5))
         printf ("%d is divisible by 5\n", i);
      ;
   }
}

void func2 (int j)
{
   int   k, m;

   k = j % 10 ? 1 : 0;
   if (k) {
      m = 23;
      k = m * m;
   }
}
                            C O D E   O F F S E T S

                              main "myfile.c"

Line   Offset      Line   Offset      Line   Offset      Line   Offset      Line   Offset
7           8      8          18      9          20

                              func1 "myfile.c"

Line   Offset      Line   Offset      Line   Offset      Line   Offset      Line   Offset
17          c      18         14      19         24      20         34



                              func2 "myfile.c"

Line   Offset      Line   Offset      Line   Offset      Line   Offset      Line   Offset
30       4     31         20     32       28     33        30
```

# A Run-Time Diagnostic Messages

This appendix lists the run-time error messages generated by HP C/XL programs. The error messages are listed in numerical order. Possible causes for each error are provided along with actions you may take to correct the error. An example of an HP C run-time error is shown below:

```
**** MISSING NAME FOR REDIRECT (CERR 4)
```

where:

| | |
|---|---|
| `MISSING NAME FOR REDIRECT` | is the message text. |
| `CERR` | indicates the subsystem name. |
| `4` | indicates the diagnostic message number. |

## Error Messages

| | |
|---|---|
| 1 | `AMBIGUOUS INPUT REDIRECT` |
| | More than one input redirection character ( < ) appears in the INFO string. For example,<br>`RUN prog; INFO="<` *`infile1`* `<` *`infile2`*`"` |
| | Remove one of the input redirection specifiers. |

| | |
|---|---|
| 2 | `AMBIGUOUS OUTPUT REDIRECT` |
| | More than one output redirection character ( > ) appears in the INFO string. For example,<br>`RUN prog; INFO=">` *`outfile1`* `>` *`outfile2`*`"` |
| | Remove one of the output redirection specifiers. |
| 3 | `BAD NAME FOR REDIRECT` |
| | The file name specified for input/output redirection in the INFO string does not begin with an asterisk ( * ), a dollar sign ( $ ), or an alphabetic character. For example,<br>`RUN prog; INFO=">` *`2badfile`*`"` |
| | Specify a valid file name for input/output redirection. |
| 4 | `MISSING NAME FOR REDIRECT` |
| | An input/output redirection character ( < or > ) appears in the INFO string but no corresponding file name is specified. For example,<br>`RUN prog; INFO="<"` |
| | Specify a file name for the redirection. |

| | |
|---|---|
| 5 | `UNMATCHED QUOTE WITHIN INFO STRING` |
| | A quoted string within the INFO string contains a beginning quote but no matching end quote. For example,<br>`RUN prog; INFO="'`*`quoted string`*`"` |
| | Add a matching end quote to the quoted string. |

| | |
|---|---|
| 6 | `REDIRECTION OF STDIN FAILED` |
| | The file specified for input redirection in the INFO string can not be opened. For example,<br>`RUN prog; INFO="<` *`badfile`*`"` |
| | where *`badfile`* does not exist. |
| | Check to see if the file specified for redirection exists and is accessible for reading by the user invoking the program. |
| 7 | `REDIRECTION OF STDOUT FAILED` |

| | |
|---|---|
| | The file specified for output redirection in the INFO string can not be opened. For example,<br>`RUN prog; INFO="> `*`badfile`*`"` |
| | where *`badfile`* can not be opened for writing. |
| | Check to see if the file specified for redirection is accessible for writing by the user invoking the program. |
| 9 | `NO FILES FOUND IN FILE-SET` |
| | A file-set wildcard specified in the INFO string did not match any files. For example,<br>`RUN prog; INFO="@.`*`nofiles`*`"` |
| | where *`nofiles`* is a group containing no files. |
| | Check to see if the file-set wildcard represents a non-empty file-set. |

| | |
|---|---|
| 10 | `ERROR IN EXPANDING FILE-SET WILDCARD` |
| | The file-set wildcard expander encountered an error in trying to expand a file-set wildcard in the INFO string. For example,<br>`RUN prog; INFO="@.`*badgroup*`"` |
| | where *badgroup* is a non-existent group. |
| | Check to see if the file-set wildcard represents a valid file-set. |
| 12 | `INFO STRING ARG LIST TOO LONG` |
| | The number of arguments (argv elements) specified in the INFO string exceeds the 1023 argument limit. For example,<br>`RUN prog; INFO="@.@.@"` |
| | where the file-set @.@.@ contains more than 1023 files. |
| | Reduce the number of arguments specified in the INFO string. |
| 13 | `AMBIGUOUS USE OF QUOTES WITHIN INFO STRING` |
| | A quote character that is not a closing quote or an escaped quote is specified within a quoted string in the INFO string. For example,<br>`RUN prog; INFO="'abcd'e"` |
| | The quote character between the characters `d` and `e` is not considered a closing quote because the character following it is not a valid string terminator, such as a space, tab or redirection character. |
| | Insert a string terminator character after the closing quote. For example,<br>`RUN prog; INFO="'abcd' e"` |

# B Syntax Summary

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

# Lexical Grammar

## Tokens

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

preprocessing-token ::=
        header-name
        identifier
        pp-number
        character-constant
        string-literal
        operator
        punctuator
        each non-white-space character cannot be one of the above
```

## Keywords

```
keyword ::= any word from the set:
        auto        extern      sizeof
        break       float       static
        case        for         struct
        char        goto        switch
        const       if          typedef
        continue    int         union
        default     long        unsigned
        do          register    void
        double      return      volatile
        else        short       while
        enum        signed
```

## 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]
    long-suffix [unsigned-suffix]

unsigned-suffix ::=
    u  U

long-suffix ::=
    l  L

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