The C Programming Language

D.1 Overview

This appendix is a C reference manual oriented toward the novice C programmer. It covers a significant portion of the language, including material not covered in the main text of this book. The intent of this appendix is to provide a quick reference to various features of the language for use during programming. Each item covered within the following sections contains a brief summary of a particular C feature and an illustrative example, when appropriate.

D.2 C Conventions

We start our coverage of the C programming language by describing the lexical elements of a C program and some of the conventions used by C programmers for writing C programs.

D.2.1 Source Files

The C programming convention is to separate programs into files of two types: source files (with the extension `.c`) and header files (with the extension `.h`). Source files, sometimes called `.c` or dot-c files, contain the C code for a group of related functions. For example, functions related to managing a stack data structure might be placed in a file named `stack.c.` Each `.c` file is compiled into an object file, and these objects are linked together into an executable image by the linker.

D.2.2 Header Files

Header files typically do not contain C statements but rather contain function, variable, structure, and type declarations, as well as preprocessor macros. The programming convention is to couple a header file with the source file in which the declared items are *defined*. For example, if the source file `stdio.c` contains the definitions for the functions `printf`, `scanf`, `getchar`, and `putchar`, then the header file `stdio.h` contains the declarations for these functions. If one of these functions is called from another `.c` file, then the `stdio.h` header file should be `#include`d to get the proper function declarations.

D.2.3 Comments

In C, comments begin with the two-character delimiter `/*` and end with `*/`. Comments can span multiple lines. Comments within comments are not legal and will generate a syntax error on most compilers. Comments within strings or character literals are not recognized as comments and will be treated as part of the character string. While some C compilers accept the C++ notation for comments (`//`), the ANSI C standard only allows for `/*` and `*/`.

D.2.4 Literals

C programs can contain literal constant values that are integers, floating point values, characters, character strings, or enumeration constants. These literals can be used as initializers for variables, or within expressions. Some examples are provided in the following subsections.

Integer

Integer literals can be expressed either in decimal, octal, or hexadecimal notation. If the literal is prefixed by a `0` (zero), it will be interpreted as an octal number. If the literal begins with a `0x`, it will be interpreted as hexadecimal (thus it can consist of the digits 0 through 9 and the characters *a* through *f*. Uppercase *A* through *F* can be used as well. An unprefixed literal (i.e., it doesn't begin with a `0` or `0x`) indicates it is in decimal notation and consists of a sequence of digits. Regardless of its base, an integer literal can be preceded by a minus sign, −, to indicate a negative value.

An integer literal can be suffixed with the letter *l* or *L* to indicate that it is of type `long int`. An integer literal suffixed with the letter *u* or *U* indicates an unsigned value. Refer to Section D.3.2 for a discussion of `long` and `unsigned` types.

The first three examples that follow express the same number, 87. The two last versions express it as an `unsigned int` value and as a `long int` value.

```
87      /*  87 in decimal     */
0x57    /*  87 in hexadecimal */
0127    /*  87 in octal       */
-24     /* -24 in decimal     */
-024    /* -20 in octal       */
-0x24   /* -36 in hexadecimal */
87U
87L
```

Floating Point

Floating point constants consist of three parts: an integer part, a decimal point, and a fractional part. The fractional part and integer part are optional, but one of the two must be present. The number preceded by a minus sign indicates a negative value. Several examples follow:

```
1.613123
.613123
1.        /* expresses the number 1.0 */
-.613123
```

Floating point literals can also be expressed in exponential notation. With this form, a floating point constant (such as 1.613123) is followed by an *e* or *E*. The *e* or *E* signals the beginning of the integer exponent, which is the power of 10 by which the part preceding the exponent is multiplied. The exponent can be a negative value. The exponent is obviously optional, and if used, then the decimal point is optional. Examples follow:

```
6.023e23      /* 6.023 * 10^23      */
454.323e-22   /* 454.323 * 10^(-22) */
5E13          /* 5.0 * 10^13         */
```

By default, a floating point type is a `double` or double-precision floating point number. This can be modified with the optional suffix *f* or *F*, which indicates a `float` or single-precision floating point number. The suffix *l* or *L* indicates a `long double` (see Section D.3.2).

Character

A character literal can be expressed by surrounding a particular character by single quotes, e.g., `'c'`. This converts the character into the internal character code used by the computer, which for most computers today, including the LC-3, is ASCII.

Table D.1 lists some special characters that typically cannot be expressed with a single keystroke. The C programming language provides a means to state them via a special sequence of characters. The last two forms, octal and hexadecimal, specify ways of stating an arbitrary character by using its code value, stated as either octal or hex. For example, the character 'S', which has the ASCII value of 83 (decimal), can be stated as '\0123' or '\x53'.

String Literals

A string literal within a C program must be enclosed within double quote characters, `"`. String literals have the type `char *` and space for them is allocated in

Table D.1	Special Characters in C
Character	Sequence
newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
formfeed	\f
audible alert	\a
backslash \	\\
question mark ?	\?
single quote '	\'
double quote "	\"
octal number	\0nnn
hexadecimal number	\xnnn

a special section of the memory address space reserved for literal constants. The termination character `'\0'` is automatically added to the character string. The following are two examples of string literals:

```
char greeting[10] = "bon jour!";
printf("This is a string literal");
```

String literals can be used to initialize character strings, or they can be used wherever an object of type `char *` is expected, for example as an argument to a function expecting a parameter of type `char *`. String literals, however, cannot be used for the assignment of arrays. For example, the following code is **not** legal in C.

```
char greeting [10];

greeting = "bon jour!";
```

Enumeration Constants

Associated with an enumerated type (see Section D.3.1) are enumerators, or enumeration constants. These constants are of type `int`, and their precise value is defined by the enumerator list of an enumeration declaration. In essence, an enumeration constant is a symbolic, integral value.

D.2.5 Formatting

C is a freely formatted language. The programmer is free to add spaces, tabs, carriage returns, new lines between and within statements and declarations. C programmers often adopt a style helpful for making the code more readable, which includes adequate indenting of control constructs, consistent alignment of open and close braces, and adequate commenting that does not obstruct someone trying to read the code. See the numerous examples in the C programming chapters of the book for a typical style of formatting C code.

D.2.6 Keywords

The following list is a set of reserved words, or keywords, that have special meaning within the C language. They are the names of the primitive types, type modifiers, control constructs, and other features natively supported by the language. These names cannot be used by the programmer as names of variables, functions, or any other object that the programmer might provide a name for.

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

D.3 Types

In C, expressions, functions, and objects have types associated with them. The type of a variable, for example, indicates something about the actual value the variable represents. For instance, if the variable `kappa` is of type `int`, then the value (which is essentially just a bit pattern) referred to by `kappa` will be interpreted as a signed integer. In C, there are the *basic data types*, which are types natively supported by the programming language, and *derived types*, which are types based on basic types and which include programmer-defined types.

D.3.1 Basic Data Types

There are several predefined basic types within the C language: `int`, `float`, `double`, and `char`. They exist automatically within all implementations of C, though their sizes and range of values depends upon the computer system being used.

int

The binary value of something of `int` type will be interpreted as a signed whole number. Typical computers use 32 bits to represent signed integers, expressed in 2's complement form. Such integers can take on values between (and including) $-2,147,483,648$ and $+2,147,483,647$.

float

Objects declared of type `float` represent single-precision floating point numbers. These numbers typically, but not always, follow the representations defined by the IEEE standard for single-precision floating point numbers, which means that the type is a 32-bit type, where 1 bit is used for sign, 8 bits for exponent (expressed in bias-127 code), and 23 bits for fraction. See Section 2.7.1.

double

Objects declared of type `double` deal with double-precision floating point numbers. Like objects of type `float`, objects of type `double` are also typically represented using the IEEE standard. The precise difference between objects of type `float` and of type `double` depends on the system being used; however, the ANSI C standard specifies that the precision of a `double` should never be less than that of a `float`. On most machines a `double` is 64 bits.

char

Objects of character type contain a single character, expressed in the character code used by the computer system. Typical computer systems use the ASCII character code (see Appendix E). The size of a `char` is large enough to store a character from the character set. C also imposes that the size of a `short int` must be at least the size of a `char`.

Collectively, the `int` and `char` types (and enumerated types) are referred to as *integral* types, whereas `float` and `double` are floating types.

Enumerated Types

C provides a way for the programmer to specify objects that take on symbolic values. For example, we may want to create a type that takes on one of four values: Penguin, Riddler, CatWoman, Joker. We can do so by using an *enumerated* type, as follows:

```
/* Specifier */
enum villains { Penguin, Riddler, CatWoman, Joker };

/* Declaration */
enum villains badGuy;
```

The variable badGuy is of the enumerated type villains. It can take on one of the four symbolic values defined by enumerator list in the specifier. The four symbolic values are called *enumeration constants* (see Section D.2.4) and are actually integer values.

In an enumerator list, the value of the first enumeration constant will be 0, the next will be 1, and so forth. In the type villains, the value of Penguin will be 0, Riddler will be 1, CatWoman will be 2, Joker will be 3. The value of an enumerator can be explicitly set by the programmer by using the assignment operator, =. For example,

```
/* Specifier */
enum villains { Penguin = 3, Riddler, CatWoman, Joker };
```

causes Penguin to be 3, Riddler to be 4, and so forth.

D.3.2 Type Qualifiers

The basic types can be modified with the use of a type qualifier. These modifiers alter the basic type in some small fashion or change its default size.

signed, unsigned

The types int and char can be modified with the use of the signed and unsigned qualifiers. By default, integers are signed; the default on characters depends on the computer system.

For example, if a computer uses 32-bit 2's complement signed integers, then a signed int can have any value in the range $-2,147,483,648$ to $+2,147,483,647$. On the same machine, an unsigned int can have a value in the range 0 to $+4,294,967,295$.

```
signed int c;      /* the signed modifier is redundant */
unsigned int d;

signed char j;     /* forces the char to be interpreted
                      as a signed value */

unsigned char k;   /* the char will be interpreted as an
                      unsigned value */
```

`long, short`

The qualifiers `long` and `short` allow the programmer to manipulate the physical size of a basic type. For example, the qualifiers can be used with an integer to create `short int` and `long int`.

It is important to note that there is no strict definition of how much larger one type of integer is than another. The C language states only that the size of a `short int` is less than or equal to the size of an `int`, which is less than or equal to the size of a `long int`. Stated more completely and precisely:

```
sizeof(char) <= sizeof(short int) <= sizeof(int) <= sizeof(long int)
```

New computers that support 64-bit data types make a distinction on the `long` qualifier. On these machines, a `long int` might be a 64-bit integer, whereas an `int` might be a 32-bit integer. The range of values of types on a particular computer can be found in the standard header file `<limits.h>`. On most UNIX systems, it will be in the `/usr/include` directory.

The following are several examples of type modifiers on the integral data types.

```
short int q;
long int p;
unsigned long int r;
```

The `long` and `short` qualifiers can also be used with the floating type `double` to create a floating point number with higher precision or larger range (if such a type is available on the computer) than a `double`. As stated by the ANSI C specification: the size of a `float` is less than or equal to the size of a `double`, which is less than or equal to the size of a `long double`.

```
double x;
long double y;
```

`const`

A value that does not change through the course of execution can be qualified with the `const` qualifier. For example,

```
const double pi = 3.14159;
```

By using this qualifier, the programmer is providing information that might enable an optimizing compiler to perform more powerful optimizations on the resulting code. All variables with a `const` qualifier must be explicitly initialized.

D.3.3 Storage Class

Memory objects in C can be of the *static* or *automatic* storage class. Objects of the automatic class are local to a block (such as a function) and lose their value once their block is completed. By default, local variables within a function are of the automatic class and are allocated on the run-time stack (see Section 14.3.1).

Objects of the *static* class retain their values throughout program execution. Global variables and other objects declared outside of all blocks are of the static class. Objects declared within a function can be qualified with the `static` qualifier to indicate that they are to be allocated with other static objects, allowing their

value to persist across invocations of the function in which they are declared. For example,

```
int Count(int x)
{
  static int y;

  y++;
  printf("This function has been called %d times.", y);
}
```

The value of `y` will not be lost when the activation record of `Count` is popped off the stack. To enable this, the compiler will allocate a static local variable in the global data section. Every call of the function `count` updates the value of `y`.

Unlike typical local variables of the automatic class, variables of the static class are initialized to zero. Variables of the automatic class must be initialized by the programmer.

There is a special qualifier called `register` that can be applied to objects in the automatic class. This qualifier provides a hint to the compiler that the value is frequently accessed within the code and should be allocated in a register to potentially enhance performance. The compiler, however, treats this only as a suggestion and can override or ignore this specifier based on its own analysis.

Functions, as well as variables, can be qualified with the qualifier `extern`. This qualifier indicates that the function's or variable's storage is defined in another object module that will be linked together with the current module when the executable is constructed.

D.3.4 Derived Types

The derived types are extensions of the basic types provided by C. The derived types include pointers, arrays, structures, and unions. Structures and unions enable the programmer to create new types that are aggregations of other types.

Arrays

An array is a sequence of objects of a particular type that is allocated sequentially in memory. That is, if the first element of the array of type T is at memory location X, the next element will be at memory location $X + $ `sizeof(T)`, and so forth. Each element of the array is accessible using an integer index, starting with the index 0. That is, the first element of array `list` is `list[0]`, numbered starting at 0. The size of the array must be stated as a constant integral expression (it is not required to be a literal) when the array is declared.

```
char string[100]; /* Declares array of 100 characters */
int  data[20];    /* Declares array of 20 integers */
```

To access a particular element within an array, an index is formed using an integral expression within square brackets, `[ ]`.

```
data[0]        /* Accesses first element of array data */
data[i + 3]    /* The variable i must be an integer */
string[x + y]  /* x and y must be integers */
```

The compiler is not required to check (nor is it required to generate code to check) whether the value of the index falls within the bounds of the array. The responsibility of ensuring proper access to the array is upon the programmer. For example, based on the previous declarations and array expressions, the reference `string[x + y]`, the value of `x + y` should be 100 or less; otherwise the reference exceeds the bounds of the array `string`.

Pointers

Pointers are objects that are addresses of other objects. Pointer types are declared by prefixing an identifier with an asterisk, `*`. The type of a pointer indicates the type of the object that the pointer points to. For example,

```
int *v;   /* v points to an integer */
```

C allows a restricted set of operations to be used on pointer variables. Pointers can be manipulated in expressions, thus allowing "pointer arithmetic" to be performed. C allows assigment between pointers of the same type, or assignment, of a pointer to 0. Assignment of a pointer to the constant value 0 causes the generation of a null pointer. Integer values can be added to or subtracted from a pointer value. Also, pointers of the same type can be compared (using the relational operators) or subtracted from one another, but this is meaningful only if the pointers involved point to elements of the same array. All other pointer manipulations are not explicitly allowed in C but can be done with the appropriate casting.

Structures

Structures enable the programmer to specify an aggregate type. That is, a structure consists of member elements, each of which has its own type. The programmer can specify a structure using the following syntax. Notice that each member element has its own type.

```
struct tag_id {
   type1 member1;
   type2 member2;
    :
    :
   typeN memberN;
};
```

This structure has member elements named `member1` of type `type1`, `member2` of `type2`, up to `memberN` of `typeN`. Member elements can take on any basic or derived type, including other programmer-defined types.

The programmer can specify an optional tag, which in this case is `tag_id`. Using the tag, the programmer can declare structure variables, such as the variable `x` in the following declaration:

```
struct tag_id x;
```

A structure is defined by its tag. Multiple structures can be declared in a program with the same member elements and member element identifiers; they are different if they have different tags.

Alternatively, variables can be declared along with the structure declaration, as shown in the following example. In this example, the variable `firstPoint` is declared along with the structure. The array `image` is declared using the structure tag `point`.

```
struct point {
  int x;
  int y;
} firstPoint;


/* declares an array of structure type variables */
struct point image[100];
```

See Section 19.2 for more information on structures.

Unions

Structures are containers that hold multiple objects of various types. Unions, on the other hand, are containers that hold a single object that can take on different predetermined types at various points in a program. For example, the following is the declaration of a union variable `joined`:

```
union u_tag {
  int    ival;
  double fval;
  char   cval;
} joined;
```

The variable `joined` ultimately contains bits. These bits can be an integer, double, or character data type, depending on what the programmer decides to put there. For example, the variable will be treated as an integer with the expression `joined.ival`, or as a double-precision floating point value with `joined.fval`, or as a character with `joined.cval`. The compiler will allocate enough space for union variables as required for the largest data type.

D.3.5 `typedef`

In C, a programmer can use `typedef` to create a synonym for an existing type. This is particularly useful for providing names for programmer-defined types. The general form for a `typedef` follows:

```
typedef type name;
```

Here, `type` can be any basic type, enumerated type, or derived type. The identifier `name` can be any legal identifier. The result of this `typedef` is that `name` is a synonym for `type`. The `typedef` declaration is an important feature for enhancing code readability; a well-chosen type name conveys additional information about the object declared of that type. Following are some examples.

```
typedef enum {coffee, tea, water, soda} Beverage;
Beverage drink;     /* Declaration uses previous typedef */
typedef struct {
   int xCoord;
   int yCoord;
   int color;
} Pixel;

Pixel bitmap[1024*820]; /*Declares an array of pixels*/
```

D.4 Declarations

An *object* is a named section of memory, such as a variable. In C, an object must be declared with a declaration before it can be used. Declarations inform the compiler of characteristics, such as its type, name, and storage class, so that correct machine code can be generated whenever the object is manipulated within the body of the program.

In C, functions are also declared before they are used. A function declaration informs the compiler about the return value, function name, and types and order of input parameters.

D.4.1 Variable Declarations

The format for a variable declaration is as follows:

```
[storage-class] [type-qualifier] {type} {identifier} [ = initializer] ;
```

The curly braces, { }, indicate items that are required and the square brackets, [ ], indicate optional items.

The optional *storage-class* can be any storage class modifier listed in Section D.3.3, such as `static`.

The optional *type-qualifier* can be any legal type qualifiers, such as the qualifiers provided in Section D.3.2.

The *type* of a variable can be any of the basic types (`int`, `char`, `float`, `double`), enumerated types, or derived type (array, pointer, structure, or union).

An *identifier* can be any sequence of letters, digits, and the underscore character, _. The first character must be a letter or the underscore character. Identifiers can have any length, but for most variables you will use, at least 31 characters will be significant. That is, variables that differ only after the 31st character might be treated as the same variable by an ANSI C compiler. Uppercase letters are different from lowercase, so the identifier `sum` is different from `Sum`. Identifiers must be different from any of the C keywords (see Section D.2.6). Several examples of legal identifiers follow. Each is a distinct identifier.

```
blue
Blue1
Blue2
_blue_
bluE
primary_colors
primaryColors
```

The *initializer* for variables of automatic storage (see Section D.3.3) can be any expression that uses previously defined values. For variables of the static class (such as global values) or external variables, the initializer must be a constant expression.

Also, multiple identifiers (and initializers) can be placed on the same line, creating multiple variables of the same type, having the same storage class and type characteristics.

```
static long unsigned int k = 10UL;
register char 1 = 'Q';
int list[100];
struct node_type n;  /* Declares a structure variable */
```

Declarations can be placed at the beginning of any *block* (see Section D.6.2), before any statements. Such declarations are visible only within the block in which they appear. Declarations can also appear at the outermost level of the program, outside of all functions. Such declarations are *global variables*. They are visible from all parts of the program. See Section 12.2.3 for more information on variable declarations.

D.4.2 Function Declarations

A function's declaration informs the compiler about the type of value returned by the function and the type, number, and order of parameters the function expects to receive from its caller. The format for a function declaration is as follows:

```
{type} {function-id}([type1] [, type2], ... [, typeN]);
```

The curly braces, { }, indicate items that are required and the square brackets, [ ], indicate items that are optional.

The *type* indicates the type of the value returned by the function and can be of any basic type, enumerated type, a structure, a union, a pointer, or `void` (note: it cannot be an array). If a function does not return a value, then its type must be declared as `void`.

The *function-id* can be any legal identifier that has not already been defined.

Enclosed within parentheses following the *function-id* are the types of each of the input parameters expected by the function, indicated by *type1*, *type2*, *typeN*, each separated by a comma. Optionally, an identifier can be supplied for each argument, indicating what the particular argument will be called within the function's definition. For example, the following might be a declaration for a function that returns the average of an array of integers:

```
int Average(int numbers[], int howMany);
```

D.5 Operators

In this section, we describe the C operators. The operators are grouped by the operations they perform.

D.5.1 Assignment Operators

C supports multiple assignment operators, the most basic of which is the simple
assigment operator `=`. All assignment operators associate from right to left.

A standard form for a simple assignment expression is as follows:

```
{left-expression} = {right-expression}
```

The *left-expression* must be a modifiable object. It cannot, for example, be a
function, an object with a type qualifier `const`, or an array (it can, however, be
an element of an array). The *left-expression* is often referred to as an *lvalue*. The
left-expression can be an object of a structure or union type.

After the assignment expression is evaluated, the value of the object referred
to by the *left-expression* will take on the value of the *right-expression*. In most
usages of the assignment operator, the types of the two expressions will be the
same. If they are different, and both are basic types, then the right operand is
converted to the type of the left operand.

The other assignment operators include:

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

All of these assignment operators combine an operation with an assignment.
In general, `A op= B` is equivalent to `A = A op (B)`. For example, `x += y` is
equivalent to `x = x + y`.

Examples of the various assignment operators can be found in Sections 12.3.2
and 12.6.4.

D.5.2 Arithmetic Operators

C supports basic arithmetic operations via the following binary operators:

```
+    -    *    /    %
```

These operators perform addition, subtraction, multiplication, division, and
modulus. These operators are most commonly used with operands of the basic
types (`int`, `double`, `float`, and `char`). If the operands have different types (such
as a floating point value plus an integer), then the resulting expression is converted
according to the conversion rules (see Section D.5.11). There is one restriction,
however: the operands of the modulus operator `%` must be of the integral type
(e.g., `int`, `char`, or enumerated).

The addition and subtraction operators can also be used with pointers that
point to values within arrays. The use of these operators in this context is referred
to as pointer arithmetic. For example, the expression `ptr + 1` where `ptr` is of
type `type *`, is equivalent to `ptr + sizeof(type)`. The expression `ptr + 1`
generates the address of the next element in the array.

C also supports the two unary operators `+` and `-`. The negation operator, `-`,
generates the negative of its operand. The unary plus operator, `+`, generates its
operand. This operator is included in the C language primarily for symmetry with
the negation operator.

For more examples involving the arithmetic operators, see Section 12.3.3.

D.5.3 Bit-Wise Operators

The following operators:

```
&    |    ^    ~    <<    >>
```

are C's bit-wise operators. They perform bit-wise operation only on integral values. That is, they cannot be used with floating point values.

The left shift operator, `<<`, and right shift operator, `>>`, evaluate to the value of the left operand shifted by the number of bit positions indicated by the right operand. In ANSI C, if the right operand is greater than the number of bits in the representation (say, for example, 33 for a 32-bit integer) or negative, then the result is undefined.

Table D.2 provides some additional details on these operators. It provides an example usage and evaluation of each with an integer operand `x` equal to 186 and the integer operand `y` equal to 6.

D.5.4 Logical Operators

The logical operators in C are particularly useful for constructing logical expressions with multiple clauses. For example, if we want to test whether both condition A and condition B are true, then we might want to use the logical AND operator.

The logical AND operator takes two operands (which do not need to be of the same type). The operator evaluates to a 1 if both operands are nonzero. It evaluates to 0 otherwise.

The logical OR operator takes two operands and evaluates to 1 if either is nonzero. If both are zero, the operator evaluates to 0.

The logical NOT operator is a unary operate that evaluates to the logical inverse of its operand: it evaluates to 1 if the operand is zero, 0 otherwise.

The logical AND and logical OR operators are *short-circuit* operators. That is, if in evaluating the left operand, the value of the operation becomes known, then the right operand is not evaluated. For example, in evaluating `(x || y++)`, if `x` is nonzero, then `y++` will not be evaluated, meaning that the side effect of the increment will not occur.

Table D.3 provides some additional details on the logical operators and provides an example usage and evaluation of each with an integer operand `x` equal to 186 and the integer operand `y` equal to 6.

Table D.2	Bit-Wise Operators in C		x=186 y=6
Operator Symbol	Operation	Example Usage	
&	bit-wise AND	x & y	2
|	bit-wise OR	x | y	190
~	bit-wise NOT	~ x	−187
^	bit-wise XOR	x ^ y	188
«	left shift	x « y	11904
»	right shift	x » y	2

Table D.3	Bit-Wise Operators in C		
			x=186
Operator Symbol	Operation	Example Usage	y=6
&&	logical AND	x && y	1
||	logical OR	x || y	1
!	logical NOT	!x	0

D.5.5 Relational Operators

The following operators:

```
==   !=   >   >=   <   <=
```

are the relational operators in C. They perform a relational comparison between the left and right operands, such as equal to, not equal to, and greater than. The typical use of these operators is to compare expressions of the basic types. If the relationship is true, then the result is the integer value 1; otherwise it is 0. Expressions of mixed type undergo the standard type conversions described in Section D.5.11. C also allows the relational operators to be used on pointers. However, such pointer expressions only have meaning if both pointers point to the same object, such as the same array.

D.5.6 Increment/Decrement Operators

The increment/decrement operators in C are `++` and `--`. They increment or decrement the operand by 1. Both operators can be used in *prefix* and *postfix* forms.

In the prefix form, for example `++x`, the value of the object is incremented (or decremented). The value of the expression is then the value of the result. For example, after the following executes:

```
int x = 4;
int y;

y = ++x;
```

both `x` and `y` equal 5.

In the postfix form, for example `x++`, the value of the expression is the value of the operand prior to the increment (or decrement). Once the value is recorded, the operand is incremented (or decremented) by 1. For example, the result of the following code:

```
int x = 4;
int y;

y = x++;
```

is that `x` equals 5 and `y` equals 4.

Like the addition and subtraction operators, the increment and decrement operators can be used with pointer types. See Section D.5.2.

D.5.7 Conditional Expression Operators

The conditional expression operator in C has the following form:

```
{expressionA} ? {expressionB} : {expressionC}
```

Here, if *expressionA* is logically true, that is, it evaluates to a nonzero value, then the value of the entire expression is the value of *expressionB*. If *expressionA* is logically false, that is, it evaluates to zero, then the value of the entire expression is the value of *expressionC*. For example, in the following code segment:

```
w = x ? y : z;
```

the value of the conditional expression `x ? y : z` will depend on the value of `x`. If `x` is nonzero, then `w` will be assigned the value of `y`. Otherwise `w` will be assigned the value of `z`.

Like the logical AND and logical OR operators, the conditional expression short-circuits the evaluation of *expressionB* or *expressionC*, depending on the state of *expressionA*. See Section D.5.4.

D.5.8 Pointer, Array, and Structure Operators

This final batch of operators performs address-related operations for use with the derived data types.

Address Operator

The address operator is the `&`. It takes the address of its operand. The operand must be a memory object, such as a variable, array element, or structure member.

Dereference Operator

The complement of the address operator is the dereference operator. It returns the object to which the operand is pointing. For example, given the following code:

```
int *p;
int x = 5;

p = &x;
*p = *p + 1;
```

the expression `*p` returns `x`. When `*p` appears on the left-hand side of an assignment operator, it is treated as an lvalue (see Section D.5.1). Otherwise `*p` evaluates to the value of `x`.

Array Reference

In C, an integral expression within square brackets, `[ ]`, designates a subscripted array reference. The typical use of this operator is with an object declared as an array. The following code contains an example of an array reference on the array `list`.

```
int x;
int list [100];

x = list [x + 10];
```

Structure and Union References

C contains two operators for referring to member elements within a structure or union. The first is the dot, or period, which directly accesses the member element of a structure or union variable. The following is an example:

```
struct pointType {
  int x;
  int y;
};
typedef pointType Point;

Point pixel;

pixel.x = 3;
pixel.y = pixel.x + 10;
```

The variable `pixel` is a structure variable, and its member elements are accessed using the dot operator.

The second means of accessing member elements of a structure is the arrow, or `->` operator. Here, a pointer to a structure or union can be dereferenced and a member element selected with a single operator. The following code demonstrates:

```
Point pixel;
Point *ptr;

ptr = &pixel;
ptr->x = ptr->x + 1;
```

Here, the pointer variable `ptr` points to the structure variable `pixel`.

D.5.9 `sizeof`

The `sizeof` operator returns the number of bytes required to store an object of the type specified. For example, `sizeof(int)` will return the number of bytes occupied by an integer. If the operand is an array, then `sizeof` will return the size of the array. The following is an example:

```
int list[45];

struct example_type {
  int    valueA;
  int    valueB;
  double valueC;
};
typedef struct example_type Example;

...

sizeA = sizeof(list);     /* 45 * sizeof(int)  */
sizeB = sizeof(Example);  /* Size of structure */
```

Table D.4   Operator Precedence, from Highest to Lowest. Descriptions of Some Operators are Provided in Parentheses

Precedence Group	Associativity	Operators
1 (highest)	l to r	`()` (function call) `[ ]` (array index) `.` `>`
2	r to l	`++` `--` (postfix versions)
3	r to l	`++` `--` (prefix versions)
4	r to l	`*` (indirection) `&` (address of)
		`+` (unary) `-` (unary) `~` `!` `sizeof`
5	r to l	`(type)` (type cast)
6	l to r	`*` (multiplication) `/` `%`
7	l to r	`+` (addition) `-` (subtraction)
8	l to r	`«` `»`
9	l to r	`<` `>` `<=` `>=`
10	l to r	`==` `!=`
11	l to r	`&`
12	l to r	`^`
13	l to r	`|`
14	l to r	`&&`
15	l to r	`||`
16	l to r	`?:`
17 (lowest)	r to l	`=` `+=` `-=` `*=` etc.

D.5.10 Order of Evaluation

The order of evaluation of an expression starts at the subexpression in the innermost parentheses, with the operator with the highest *precedence*, moving to the operator with the lowest precedence within the same subexpression. If two operators have the same precedence (for example, two of the same operators, as in the expression `2 + 3 + 4`), then the *associativity* of the operators determines the order of evaluation, either from left to right or from right to left. The evaluation of the expression continues recursively from there.

Table D.4 provides the *precedence* and *associativity* of the C operators. The operators of highest precedence are listed at the top of the table, in lower numbered precedence groups.

D.5.11 Type Conversions

Consider the following expression involving the operator *op*.

```
A op B
```

The resulting value of this expression will have a particular type associated with it. This resulting type depends on (1) the types of the operands *A* and *B*, and (2) the nature of the operator *op*.

If the types of *A* and *B* are the same and the operator can operate on that type, the result is the type defined by the operator.

When an expression contains variables that are a mixture of the basic types, C performs a set of standard arithmetic conversions of the operand values. In general, smaller types are converted into larger types, and integral types are converted into floating types. For example, if *A* is of type `double` and *B* is of type `int`, the

result is of type `double`. Integral values, such as `char`, `int`, or an enumerated type, are converted to `int` (or `unsigned int`, depending on the implementation). The following are examples.

```
char   i;
int    j;
float  x;
double y;

i * j    /* This expression is an integer */
j + 1    /* This expression is an integer */
j + 1.0  /* This expression is a float    */
i + 1.0  /* This expression is a float    */
x + y    /* This expression is a double   */
i + j + x + y          /* This is a double */
```

As in case (2) above, some operators require operands of a particular type or generate results of a particular type. For example, the modulus operator `%` only operates on integral values. Here integral type conversions are performed on the operands (e.g., `char` is converted to `int`). Floating point values are not allowed and will generate compilation errors.

If a floating point type is converted to an integral type (which does not happen with the usual type conversion, but can happen with casting as described in the next subsection), the fractional portion is discarded. If the resulting integer cannot be represented by the integral type, the result is undefined.

Casting

The programmer can explicitly control the type conversion process by *type casting*. A cast has the general form:

```
(new-type) expression
```

Here the expression is converted into the *new-type* using the usual conversion rules described in the preceding paragraphs. Continuing with the previous example code:

```
j = (int) x + y; /* This results in conversion of
                    double into an integer */
```

D.6 Expressions and Statements

In C, the work performed by a program is described by the expressions and statements within the bodies of functions.

D.6.1 Expressions

An expression is any legal combination of constants, variables, operators, and function calls that evaluates to a value of a particular type. The order of evaluation is based on the precedence and associativity rules described in Section D.5.10.

The type of an expression is based on the individual elements of the expression, according to the C type promotion rules (see Section D.5.11). If all the elements of an expression are `int` types, then the expression is of `int` type. Following are several examples of expressions:

```
a * a + b * b
a++ - c / 3
a <= 4
q || integrate(x)
```

D.6.2 Statements

In C, simple statements are expressions terminated by a semicolon, `;`. Typically, statements modify a variable or have some other side effect when the expression is evaluated. Once a statement has completed execution, the next statement in sequential order is executed. If the statement is the last statement in its function, then the function terminates.

```
c = a * a + b * b;  /* Two simple statements */
b = a++ - c / 3;
```

Related statements can be grouped togethered into a compound statement, or *block*, by surrounding them with curly braces, { }. Syntactically, the compound statement is the same as a simple statement, and they can be used interchangeably.

```
{                        /* One compound statement */
 c = a * a + b * b;
 b = a++ - c / 3;
}
```

D.7 Control

The control constructs in C enable the programmer to alter the sequential execution of statements with statements that execute conditionally or iteratively.

D.7.1 If

An `if` statement has the format

```
if  (expression)
    statement
```

If the *expression*, which can be of any basic, enumerated, or pointer types, evaluates to a nonzero value, then the *statement*, which can be a simple or compound statement, is executed.

```
if (x < 0)
   a = b + c; /* Executes if x is less than zero */
```

See Section 13.2.1 for more examples of `if` statements.

D.7.2 `If-else`

An `if-else` statement has the format

```
if (expression)
   statement1
else
   statement2
```

If the *expression*, which can be of any basic, enumerated, or pointer type, eval-uates to a nonzero value, then *statement1* is executed. Otherwise, *statement2* is executed. Both *statement1* and *statement2* can be simple or compound statements.

```
if  (x < 0)
    a = b + c; /* Executes if x is less than zero */
else
    a = b - c; /* Otherwise, this is executed.  */
```

See Section 13.2.2 for more examples of `if-else` statements.

D.7.3 `Switch`

A `switch` statement has the following format:

```
switch(expression)  {
case const-expr1:
     statement1A
     statement1B
     :

case const-expr2:
     statement2A
     statement2B
     :

     :
     :

case const-exprN:
     statementNA
     statementNB
     :

}
```

A `switch` statement is composed of an *expression*, which must be of integral type (see Section D.3.1), followed by a compound statement (though it is not required to be compound, it almost always is). Within the compound statement exist one or more `case` labels, each with an associated constant integral expression, called *const-expr1*, *const-expr2*, *const-exprN* in the preceding example. Within a `switch`, each `case` label must be different.

When a `switch` is encountered, the controlling *expression* is evaluated. If one of the case labels matches the value of *expression*, then control jumps to the statement that follows and proceeds from there.

The special case label `default` can be used to catch the situation where none of the other case labels match. If the `default` case is not present and none of the labels match the value of the controlling expression, then no statements within the `switch` are executed.

The following is an example of a code segment that uses a `switch` statement. The use of the `break` statement causes control to leave the `switch`. See Section D.7.7 for more information on `break`.

```
char k;

k = getchar();
switch (k)  {
case '+':
  a = b + c;
  break;       /* break causes control to leave switch */

case '-':
  a = b - c;
  break;

case '*':
  a = b * c;
  break;

case '/':
  a = b / c;
  break;
}
```

See Section 13.5.1 for more examples of `switch` statements.

D.7.4 `While`

A `while` statement has the following format:

```
while (expression)
  statement
```

The `while` statement is an iteration construct. If the value of *expression* evaluates to nonzero, then the *statement* is executed. Control does not pass to the subsequent statement, but rather the *expression* is evaluated again and the process is repeated. This continues until *expression* evaluates to 0, in which case control passes to the next statement. The *statement* can be a simple or compound statement.

In the following example, the `while` loop will iterate 100 times.

```
x = 0;
while  (x < 100)  {
   printf("x = %d\n", x);
   x = x + 1;
}
```

See Section 13.3.1 for more examples of `while` statements.

D.7.5 `For`

A `for` statement has the following format:

```
for (initializer;  term-expr; reinitializer)
   statement
```

The `for` statement is an iteration construct. The *initializer*, which is an expression, is evaluated only once, before the loop begins. The *term-expr* is an expression that is evaluated before each iteration of the loop. If the *term-expr* evaluates to nonzero, the loop progresses; otherwise the loop terminates and control passes to the statement following the loop. Each iteration of the loop consists of the execution of the *statement*, which makes up the body of the loop, and the evaluation of the *reinitializer* expression.

The following example is a `for` loop that iterates 100 times.

```
for (x = 0; x < 100; X++)   {
   printf("x = %d\n", x);
}
```

See Section 13.3.2 for more examples of `for` statements.

D.7.6 `Do-while`

A `do-while` statement has the format

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

The `do-while` statement is an iteration construct similar to the `while` statement. When a `do-while` is first encountered, the *statement* that makes up the loop body is executed first, then the *expression* is evaluated to determine whether to execute another iteration. If it is nonzero, then another iteration is executed (in other words, *statement* is executed again). In this manner, a `do-while` always executes its loop body at least once.

The following `do-while` loop iterates 100 times.

```
x = 0;
do {
   printf("x = %d\n", x);
   x = x + 1;
}
while (x < 100);
```

See Section 13.3.3 for more examples of `do-while` statements.

D.7.7 `Break`

A `break` statement has the format:

```
break;
```

The `break` statement can only be used in an iteration statement or in a `switch` statement. It passes control out of the smallest statement containing it to the statement immediately following. Typically, `break` is used to exit a loop before the terminating condition is encountered.

In the following example, the execution of the `break` statement causes control to pass out of the `for` loop.

```
for (x = 0; x < 100; x++)  {
  :
  :
  if (error)
      break;
  :
  :
}
```

See Section 13.5.2 for more examples of `break` statements.

D.7.8 `continue`

A `continue` statement has the following format:

```
continue;
```

The `continue` statement can be used only in an iteration statement. It prematurely terminates the execution of the loop body. That is, it terminates the current iteration of the loop. The looping expression is evaluated to determine whether another iteration should be performed. In a `for` loop the *reinitializer* is also evaluated.

If the `continue` statement is executed, then x is incremented, and the *reinitializer* executed, and the loop expression evaluated to determine if another iteration should be executed.

```
for (x = 0; x < 100; x++)  {
  :
  :
  if (skip)
      continue;
  :
  :
}
```

See Section 13.5.2 for more examples of `continue` statements.

D.7.9 `return`

A `return` statement has the format

```
return expression;
```

The `return` statement causes control to return to the current caller function, that is, the function that called the function that contains the `return` statement. Also, after the last statement of a function is executed, an implicit return is made to the caller.

The `expression` that follows the `return` is the return value generated by the function. It is converted to the return type of the function. If a function returns a value, and yet no `return` statement within the function explicitly generates a return value, then the return value is undefined.

```
return x + y;
```

D.8 The C Preprocessor

The C programming language includes a preprocessing step that modifies, in a programmer-controlled manner, the source code presented to the compiler. The most frequently used features of the C preprocessor are its macro substitution facility (`#define`), which replaces a sequence of source text with another sequence, and the file inclusion facility (`#include`), which includes the contents of a file into the source text. Both of these are described in the following subsections.

None of the preprocessor directives are required to end with a semicolon. Since `#define` and `#include` are preprocessor directives and not C statements, they are not required to be terminated by semicolons.

D.8.1 Macro Substitution

The `#define` preprocessor directive instructs the C preprocessor to replace occurrences of one character sequence with another. Consider the following example:

```
#define A B
```

Here, any token that matches A will be replaced by B. That is, the *macro* A gets *substituted* with B. The character A must appear as an individual sequence, i.e., the A in APPLE will not be substituted, and not appear in quoted strings, i.e., neither will "A".

The replacement text spans until the end of the line. If a longer sequence is required, the backslash character, \, can be used to continue to the next line.

Macros can also take arguments. They are specified in parentheses immediately after the text to be replaced. For example:

```
#define REMAINDER(X, Y)    ((X) % (Y))
```

Here, every occurrence of the macro `COPY` in the source code will be accompanied by two values, as in the following example.

```
valueC = REMAINDER(valueA, valueB + 15);
```

The macro `REMAINDER` will be replaced by the preprocessor with the replacement text provided in the `#define`, and the two arguments A and B will be substituted with the two arguments that appear in the source code. The previous code will be modified to the following after preprocessing:

```
valueC = ((valueA) % (valueB + 15));
```

Notice that the parentheses surrounding X and Y in the macro definition were required. Without them, the macro `REMAINDER` would have calculated the wrong value.

While the `REMAINDER` macro appears to be similar to a function call, notice that it incurs none of the function call overhead associated with regular functions.

D.8.2 File Inclusion

The `#include` directive instructs the preprocessor to insert the contents of a file into the source file. Typically, the `#include` directive is used to attach header files to C source files. *C header files* typically contain `#defines` and declarations that are useful among multiple source files.

There are two variations of the `#include` directive:

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

The first variation uses angle brackets, < >, around the filename. This tells the preprocessor that the header file can be found in a predefined directory, usually determined by the configuration of the system and which contains many system-related and library-related header files, such as `stdio.h`. The second variation, using double quotes, " ", around the filename, instructs the preprocessor that the header file can be found in the same directory as the C source file.

D.9 Some Standard Library Functions

The ANSI C standard library contains over 150 functions that perform a variety of useful tasks (for example, I/O and dynamic memory allocation) on behalf of your program. Every installation of ANSI C will have these functions available, so even if you make use of these functions, your program will still be portable from one ANSI C platform to another. In this section, we will describe some useful standard library functions.

へ

D.9.1 I/O Functions

The `<stdio.h>` header file must be included in any source file that contains calls to the standard I/O functions. Following is a small sample of these functions.

getchar

This function has the following declaration:

```
int getchar(void);
```

The function `getchar` reads the next character from the standard input device, or `stdin`. The value of this character is returned (as an integer) as the return value.

The behavior of `getchar` is very similar to the LC-3 input TRAP (except no input banner is displayed on the screen).

Most computer systems will implement `getchar` using buffered I/O. This means that keystrokes (assuming standard input is coming from the keyboard) will be buffered by the operating system until the Enter key is pressed. Once Enter is pressed, the entire line of characters is added to the standard input stream.

putchar

This function has the following declaration:

```
void putchar(int c);
```

The function `putchar` takes an integer value representing an ASCII character and puts the character to the standard output stream. This is similar to the LC-3 TRAP OUT.

If the standard output stream is the monitor, the character will appear on the screen. However, since many systems buffer the output stream, the character may not appear until the system's output buffer is *flushed*, which is usually done once a newline appears in the output stream.

scanf

This function has the following declaration:

```
int scanf(const char *formatstring, *ptr1, ...);
```

The function `scanf` is passed a format string (which is passed as pointer to the initial character) and a list of pointers. The format string contains format specifications that control how `scanf` will interpret fields in the input stream. For example, the specification `%d` causes `scanf` to interpret the next sequence of non–white space characters as a decimal number. This decimal is converted from ASCII into an integer value and assigned to the variable pointed to by the next pointer in the parameter list. Table D.5 contains a listing of the possible specifications for use with `scanf`. The number of pointers that follow the format string in the parameter list should correspond to the number of format specifications in the format string. The value returned by `scanf` corresponds to the number of variables that were successfully assigned.

Table D.5	scanf **Conversion Specifications**
scanf Conversions	Parameter Type
%d	signed decimal
%i	decimal, octal (leading 0), hex (leading 0x or 0X)
%o	octal
%x	hexadecimal
%u	unsigned decimal
%c	char
%s	string of non–white space characters, \0 added
%f, %e	floating point number

Table D.6	printf **Conversion Specifications**
printf Conversions	Printed as
%d, %i	signed decimal
%o	octal
%x, %X	hexadecimal (a–f or A–F)
%u	unsigned decimal
%c	single char
%s	string, terminated by \0
%f	floating point in decimal notation
%e, %E	floating point in exponential notation
%p	pointer

printf

This function has the following declaration:

```
int printf(const char *formatString, ...);
```

The function printf writes the format string (passed as a pointer to the initial character) to the standard output stream. If the format string contains a format specification, then printf will interpret the next parameter in the parameter list as indicated by the specification, and embed the interpreted value into the output stream. For example, the format specification %d will cause printf to interpret the next parameter as a decimal value. printf will write the resulting digits into the output stream. Table D.6 contains a listing of the format specifications for use with printf. In general, the number of values following the format string on the parameter list should correspond to the number of format specifications in the format string. printf returns the number of characters written to the output stream. However, if an error occurs, a negative value is returned.

D.9.2 String Functions

The C standard library contains around 15 functions that perform operations on strings (that is, null-terminated arrays of characters). To use the string functions from within a program, include the <string.h> header file in each source file that contains a call to a library string function. In this section, we describe two examples of C string functions.

strcmp

This function has the following declaration:

```
int strcmp(char *stringA, char *stringB);
```

This function compares `stringA` with `stringB`. It returns a 0 if they are equal. It returns a value greater than 0 if `stringA` is lexicographically greater than `stringB` (lexicographically greater means that `stringA` occurs later in a dictionary than `stringB`). It returns a value less than 0 if `stringA` is lexicographically less than `stringB`.

strcpy

This function has the following declaration:

```
char *strcpy(char *stringA, char *stringB);
```

This function copies `stringB` to `stringA`. It copies every character in `stringB` up to and including the null character. The function returns a pointer to `stringA` if no errors occurred.

D.9.3 Math Functions

The C standard math functions perform commonly used mathematical operations. Using them requires including the `<math.h>` header file. In this section, we list a small sample of C math functions. Each of the listed functions takes as parameters values of type `double`, and each returns a value of type `double`.

```
double sin(double x);   /* sine of x, expressed in radians   */
double cos(double x);   /* cosine of x, expressed in radians */
double tan(double x);   /* tan of x, expressed in radians    */
double exp(double x);   /* exponential function, e^x         */
double log(double x);   /* natural log of x                  */
double sqrt(double x);  /* square root of x                  */
double pow(double x, double y)   /* x^y -- x to the y power  */
```

D.9.4 Utility Functions

The C library contains a set of functions that perform useful tasks such as memory allocation, data conversion, sorting, and other miscellaneous things. The common header file for these functions is `<stdlib.h>`.

malloc

As described in Section 19.3, the function `malloc` allocates a fixed-sized chunk from memory.

This function has the following declaration:

```
void *malloc(size_t size);
```

The input parameter is the number of bytes to be allocated. The parameter is of type `size_t`, which is the same type returned by the `sizeof` operator (very often, this type is `typedef`ed as an unsigned integer). If the memory allocation

goes successfully, a pointer to the allocated region of memory is returned. If the request cannot be satisfied, the value NULL is returned.

free

This function has the following declaration:

```
void free(void *ptr);
```

This function returns to the heap a previously allocated chunk of memory pointed to by the parameter. In other words, free deallocates memory pointed to by ptr. The value passed to free must be a pointer to a previously allocated region of memory, otherwise errors could occur.

rand and srand

The C standard utility functions contain a function to generate a sequence of random numbers. The function is called rand. It does not generate a truly random sequence, however. Instead, it generates the same sequence of varying values based on an initial *seed* value. When the seed is changed, a different sequence is generated. For example, when seeded with the value 10, the generator will always generate the same sequence of numbers. However, this sequence will be different than the sequence generated by another seed value.

The function rand has the following declaration:

```
int rand(void)
```

It returns a pseudo-random integer in the range 0 to RAND_MAX, which is at least 32,767.

To seed the pseudo-random number generator, use the function srand. This function has the following declaration:

```
void srand(unsigned int seed);
```