A

C Language Summary

THIS SECTION SUMMARIZES THE C LANGUAGE in a format suitable for quick reference. It is not intended that this section be a complete definition of the language, but rather a more informal description of its features. You should thoroughly read the material in this section after you have completed the text. Doing so not only reinforces the material you have learned, but also provides you with a better global understanding of C.

This summary is based on the ANSI C99 (ISO/IEC 9899:1999) standard.

1.0 Digraphs and Identifiers

1.1 Digraph Characters

Table A.1 lists special two-character sequences (digraphs) that are equivalent to the listed single-character punctuators.

Table A.1 **Digraph Characters**

Digraph	Meaning
`<:`	`[`
`:>`	`]`
`<%`	`{`
`%>`	`}`
`%:`	`#`
`%:%:`	`##`

1.2 Identifiers

An *identifier* in C consists of a sequence of letters (upper- or lowercase), *universal character names* (Section 1.2.1), digits, or underscore characters. The first character of an identifier

must be a letter, underscore, or a universal character name. The first 31 characters of an identifier are guaranteed to be significant in an external name, and the first 63 characters are guaranteed to be significant for an internal identifier or macro name.

1.2.1 Universal Character Names

A universal character name is formed by the characters `\u` followed by four hexadecimal numbers or the characters `\U` followed by eight hexadecimal numbers. If the first character of an identifier is specified by a universal character, its value cannot be that of a digit character. Universal characters, when used in identifier names can also not specify a character whose value is less than $A0_{16}$ (other than 24_{16}, 40_{16}, or 60_{16}) or a character in the range $D800_{16}$ through $DFFF_{16}$, inclusive.

Universal character names can be used in identifier names, character constants, and character strings.

1.2.2 Keywords

The identifiers listed in Table A.2 are keywords that have a special meaning to the C compiler.

Table A.2 **Keywords**

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

2.0 Comments

You can insert comments into a program in two ways. A comment can begin with the two characters `//`. Any characters that follow on the line are ignored by the compiler.

A comment can also begin with the two characters `/*` and end when the characters `*/` are encountered. Any characters can be included inside the comment, which can extend over multiple lines of the program. A comment can be used anywhere in the program where a blank space is allowed. Comments, however, cannot be nested, which means that the first `*/` characters encountered end the comment, no matter how many `/*` characters you use.

3.0 Constants

3.1 Integer Constants

An integer constant is a sequence of digits, optionally preceded by a plus or minus sign. If the first digit is `0`, the integer is taken as an octal constant, in which case all digits that follow must be from `0` to `7`. If the first digit is `0` and is immediately followed by the letter `x` (or `X`), the integer is taken as a hexadecimal constant, and the digits that follow can be in the range from `0` to `9` or from `a` to `f` (or from `A` to `F`).

The suffix letter `l` or `L` can be added to the end of a decimal integer constant to make it a `long int` constant. If the value can't fit into a `long int`, it's treated as a `long long int`. If the suffix letter `l` or `L` is added to the end of an octal or a hexadecimal constant, it is taken as a `long int` if it can fit; if it cannot fit, it is taken as a `long long int`. Finally, if it cannot fit in a `long long int`, it is taken as an `unsigned long long int` constant.

The suffix letters `ll` or `LL` can be added to the end of a decimal integer constant to make it a `long long int`. When added to the end of an octal or a hexadecimal constant, it is taken as a `long long int` first, and if it cannot fit there, it is taken as an `unsigned long long int` constant.

The suffix `u` or `U` can be added to the end of an integer constant to make it `unsigned`. If the constant is too large to fit inside an `unsigned int`, it's taken as an `unsigned long int`. If it's too large for an `unsigned long int`, it's taken as an `unsigned long long int`.

Both an `unsigned` and a `long` suffix can be added to an integer constant to make it an `unsigned long int`. If the constant is too large to fit in an `unsigned long int`, it's taken as an `unsigned long long int`.

Both an `unsigned` and a `long long` suffix can be added to an integer constant to make it an `unsigned long long int`.

If an unsuffixed decimal integer constant is too large to fit into a `signed int`, it is treated as a `long int`. If it's too large to fit into a `long int`, it's treated as a `long long int`.

If an unsuffixed octal or hexadecimal integer constant is too large to fit into a `signed int`, it is treated as an `unsigned int`. If it's too large to fit into an `unsigned int`, it's treated as a `long int`, and if it's too large to fit into a `long int`, it's treated as an `unsigned long int`. If it's too large for an `unsigned long int`, it's taken as a `long long int`. Finally, if it's too large to fit into a `long long int`, the constant is treated as an `unsigned long long int`.

3.2 Floating-Point Constants

A floating-point constant consists of a sequence of decimal digits, a decimal point, and another sequence of decimal digits. A minus sign can precede the value to denote a negative value. Either the sequence of digits before the decimal point or after the decimal point can be omitted, but not both.

If the floating-point constant is immediately followed by the letter `e` (or `E`) and an optionally signed integer, the constant is expressed in scientific notation. This integer (the *exponent*) represents the power of 10 by which the value preceding the letter `e` (the *mantissa*) is multiplied (for example, `1.5e-2` represents 1.5×10^{-2} or .015).

A *hexadecimal* floating constant consists of a leading `0x` or `0X`, followed by one or more decimal or hexadecimal digits, followed by a `p` or `P`, followed by an optionally signed binary exponent. For example, 0x3p10 represents the value 3×2^{10}.

Floating-point constants are treated as `double` precision values by the compiler. The suffix letter `f` or `F` can be added to specify a `float` constant instead of a `double` constant. The suffix letter `l` or `L` can be added to specify a `long double` constant.

3.3 Character Constants

A character enclosed within single quotation marks is a character constant. How the inclusion of more than one character inside the single quotation marks is handled is implementation-defined. A universal character (Section 1.2.1) can be used in a character constant to specify a character not included in the standard character set.

3.3.1 Escape Sequences

Special escape sequences are recognized and are introduced by the backslash character. These escape sequences are listed in Table A.3.

Table A.3 **Special Escape Sequences**

Character	Meaning
`\a`	Audible alert
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\"`	Double quote
`\'`	Single quote
`\?`	Question mark
`\`*nnn*	Octal character value
`\u`*nnnn*	Universal character name
`\U`*nnnnnnnn*	Universal character name
`\x`*nn*	Hexadecimal character value

In the octal character case, from one to three octal digits can be specified. In the last three cases, hexadecimal digits are used.

3.3.2 Wide Character Constants

A *wide character constant* is written as `L'x'`. The type of such a constant is `wchar_t`, as defined in the standard header file `<stddef.h>`. Wide character constants provide a way to express a character from a character set that cannot be fully represented with the normal `char` type.

3.4 Character String Constants

A sequence of zero or more characters enclosed within double quotation marks represents a character string constant. Any valid character can be included in the string, including any of the escape characters listed previously. The compiler automatically inserts a null character (`'\0'`) at the end of the string.

Normally, the compiler produces a pointer to the first character in the string and the type is "pointer to `char`." However, when the string constant is used with the `sizeof` operator to initialize a character array, or with the `&` operator, the type of the string constant is "array of `char`."

Character string constants cannot be modified by the program.

3.4.1 Character String Concatenation

The preprocessor automatically concatenates adjacent character string constants together. The strings can be separated by zero or more whitespace characters. So, the following three strings

```
"a" " character "
    "string"
```

are equivalent to the single string

```
"a character string"
```

after concatenation.

3.4.2 Multibyte Characters

Implementation-defined sequences of characters can be used to *shift* back and forth between different states in a character string so that multibyte characters can be included.

3.4.3 Wide Character String Constants

Character string constants from an extended character set are expressed using the format `L"..."`. The type of such a constant is "pointer to `wchar_t`," where `wchar_t` is defined in `<stddef.h>`.

3.5 Enumeration Constants

An identifier that has been declared as a value for an enumerated type is taken as a constant of that particular type and is otherwise treated as type `int` by the compiler.

4.0 Data Types and Declarations

This section summarizes the basic data types, derived data types, enumerated data types, and `typedef`. Also summarized in this section is the format for declaring variables.

4.1 Declarations

When defining a particular structure, union, enumerated data type, or `typedef`, the compiler does not automatically reserve any storage. The definition merely tells the compiler about the particular data type and (optionally) associates a name with it. Such a definition can be made either inside or outside a function. In the former case, only the function knows of its existence; in the latter case, it is known throughout the remainder of the file.

After the definition has been made, variables can be declared to be of that particular data type. A variable that is declared to be of *any* data type *does* have storage reserved for it, unless it is an `extern` declaration, in which case it might or might not have storage allocated (see Section 6.0).

The language also enables storage to be allocated at the same time that a particular structure, union, or enumerated data type is defined. This is done by simply listing the variables before the terminating semicolon of the definition.

4.2 Basic Data Types

The basic C data types are summarized in Table A.4. A variable can be declared to be of a particular basic data type using the following format:

```
type  name = initial_value;
```

The assignment of an initial value to the variable is optional, and is subject to the rules summarized in Section 6.2. More than one variable can be declared at once using the following general format:

```
type  name = initial_value, name = initial_value, ... ;
```

Before the type declaration, an optional storage class might also be specified, as summarized in Section 6.2. If a storage class is specified, and the type of the variable is `int`, then `int` can be omitted. For example,

```
static  counter;
```

declares `counter` to be a `static int` variable.

Table A.4 **Summary of Basic Data Types**

Type	Meaning
`int`	Integer value; that is, a value that contains no decimal point; guaranteed to contain at least 16 bits of precision.
`short int`	Integer value of reduced precision; takes half as much memory as an `int` on some machines; guaranteed to contain at least 16 bits of precision.
`long int`	Integer value of extended precision; guaranteed to contain at least 32 bits of precision.
`long long int`	Integer value of extraextended precision; guaranteed to contain at least 64 bits of precision.
`unsigned int`	Positive integer value; can store positive values up to twice as large as an `int`; guaranteed to contain at least 16 bits of precision.
`float`	Floating-point value; that is, a value that can contain decimal places; guaranteed to contain at least six digits of precision.
`double`	Extended accuracy floating-point value; guaranteed to contain at least 10 digits of precision.
`long double`	Extraextended accuracy floating-point value; guaranteed to contain at least 10 digits of precision.
`char`	Single character value; on some systems, sign extension might occur when used in an expression.
`unsigned char`	Same as `char`, except ensures that sign extension does not occur as a result of integral promotion.
`signed char`	Same as `char`, except ensures that sign extension does occur as a result of integral promotion.
`_Bool`	Boolean type; large enough to store the values `0` or `1`.
`float _Complex`	Complex number.
`double _Complex`	Extended accuracy complex number.
`long double _Complex`	Extraextended accuracy complex number.
`void`	No type; used to ensure that a function that does not return a value is not used as if it does return one, or to explicitly "discard" the results of an expression. Also used as a generic pointer type (`void *`).

Note that the `signed` modifier can also be placed in front of the `short int`, `int`, `long int`, and `long long int` types. Because these types are signed by default anyway, this has no effect.

`_Complex` and `_Imaginary` data types enable complex and imaginary numbers to be declared and manipulated, with functions in the library for supporting arithmetic on these types. Normally, you should include the file `<complex.h>` in your program, which

defines macros and declares functions for working with complex and imaginary numbers. For example, a `double_Complex` variable `c1` can be declared and initialized to the value `5 + 10.5i` with a statement such as:

```
double _Complex  c1 = 5 + 10.5 * I;
```

Library routines such as `creal` and `cimag` can then be used to extract the real and imaginary part of `c1`, respectively.

An implementation is not required to support types `_Complex` and `_Imaginary`, or can optionally support one but not the other.

The header file `<stdbool.h>` can be included in a program to make working with Boolean variables easier. In that file, the macros `bool`, `true`, and `false` are defined, enabling you to write statements such as:

```
bool  endOfData = false;
```

4.3 Derived Data Types

A derived data type is one that is built up from one or more of the basic data types. Derived data types are arrays, structures, unions, and pointers. A function that returns a value of a specified type is also considered a derived data type. Each of these, with the exception of functions, is summarized in the following sections. Functions are separately covered in Section 7.0.

4.3.1 Arrays

Single-Dimensional Arrays

Arrays can be defined to contain any basic data type or any derived data type. Arrays of functions are not permitted (although arrays of function pointers are).

The declaration of an array has the following basic format:

```
type  name[n]    = { initExpression, initExpression, ... };
```

The expression *n* determines the number of elements in the array *name* and can be omitted provided a list of initial values is specified. In such a case, the size of the array is determined based on the number of initial values listed or on the largest index element referenced if *designated initializers* are used.

Each initial value must be a constant expression if a global array is defined. There can be fewer values in the initialization list than there are elements in the array, but there cannot be more. If fewer values are specified, only that many elements of the array are initialized. The remaining elements are set to `0`.

A special case of array initialization occurs in the event of character arrays, which can be initialized by a constant character string. For example,

```
char today[] = "Monday";
```

declares `today` as an array of characters. This array is initialized to the characters `'M'`, `'o'`, `'n'`, `'d'`, `'a'`, `'y'`, and `'\0'`, respectively.

If you explicitly dimension the character array and don't leave room for the terminating null, the compiler does not place a null at the end of the array:

```
char today[6] = "Monday";
```

This declares `today` as an array of six characters and sets its elements to the characters `'M'`, `'o'`, `'n'`, `'d'`, `'a'`, and `'y'`, respectively.

By enclosing an element number in a pair of brackets, specific array elements can be initialized in any order. For example

```
int     x = 1233;
int     a[] =  { [9] = x + 1, [3] = 3, [2] = 2, [1] = 1 };
```

defines a 10-element array called `a` (based on the highest index into the array), and initializes the last element to the value of `x + 1` (1234), and the first three elements to `1`, `2`, and `3`, respectively.

4.3.1.1 Variable-Length Arrays

Inside a function or block, you can dimension an array using an expression containing variables. In that case, the size is calculated at runtime. For example, the function

```
int makeVals (int n)
{
    int valArray[n];
    …
}
```

defines an automatic array called `valArray` with a size of `n` elements, where `n` is evaluated at runtime, and might vary between function calls. Variable-length arrays cannot be initialized at the time they are declared.

4.3.1.2 Multidimensional Arrays

The general format for declaring a multidimensional array is as follows:

```
type  name[d1][d2]...[dn] = initializationList;
```

The array *name* is defined to contain *d1 x d2 x … x dn* elements of the specified *type*. For example,

```
int  three_d [5][2][20];
```

defines a three-dimensional array, `three_d`, containing 200 integers.

A particular element is referenced from a multidimensional array by enclosing the desired subscript for each dimension in its own set of brackets. For example, the statement

```
three_d [4][0][15] = 100;
```

stores 100 in the indicated element of the array `three_d`.

Multidimensional arrays can be initialized in the same manner as one-dimensional arrays. Nested pairs of braces can be used to control the assignment of values to the elements in the array.

The following declares `matrix` to be a two-dimensional array containing four rows and three columns:

```
int matrix[4][3] =
        {  { 1, 2, 3 },
           { 4, 5, 6 },
           { 7, 8, 9 } };
```

Elements in the first row of `matrix` are set to the values `1`, `2`, and `3`, respectively; elements in the second row are set to the values `4`, `5`, and `6`, respectively; and in the third row, elements are set to the values `7`, `8`, and `9`, respectively. The elements in the fourth row are set to `0` because no values are specified for that row. The declaration

```
static int matrix[4][3] =
          { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

initializes `matrix` to the same values because the elements of a multidimensional array are initialized in "dimension-order"; that is, from leftmost to rightmost dimension.

The declaration

```
int matrix[4][3] =
         {  { 1 },
            { 4 },
            { 7 } };
```

sets the first element of the first row of `matrix` to `1`, the first element of the second row to `4`, and the first element of the third row to `7`. All remaining elements are set to `0` by default.

Finally, the declaration

```
int matrix[4][3] = {  [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

initializes the indicated elements of the matrix to the specified values.

4.3.2 Structures

The general format for declaring a structure is as follows:

```
struct name
{
     memberDeclaration
     memberDeclaration
       ...
}  variableList;
```

The structure *name* is defined to contain the members as specified by each *memberDeclaration*. Each such declaration consists of a type specification followed by a list of one or more member names.

Variables can be declared at the time that the structure is defined simply by listing them before the terminating semicolon, or they can subsequently be declared using the format

```
struct name  variableList;
```

This format cannot be used if *name* is omitted when the structure is defined. In that case, all variables of that structure type must be declared with the definition.

The format for initializing a structure variable is similar to that for arrays. Its members can be initialized by enclosing the list of initial values in a pair of curly braces. Each value in the list must be a constant expression if a global structure is initialized.

The declaration

```
struct  point
{
    float x;
    float y;
}  start = {100.0, 200.0};
```

defines a structure called `point` and a `struct point` variable called `start` with initial values as specified. Specific members can be designated for initialization in any order with the notation

```
.member = value
```

in the initialization list, as in

```
struct point  end = { .y = 500, .x = 200 };
```

The declaration

```
struct  entry
{
   char  *word;
   char  *def;
} dictionary[1000] = {
    { "a",       "first letter of the alphabet" },
    { "aardvark",  "a burrowing African mammal" },
    { "aback",     "to startle"                 }
};
```

declares `dictionary` to contain 1,000 `entry` structures, with the first three elements initialized to the specified character string pointers. Using designated initializers, you could have also written it like this:

```
struct  entry
{
   char  *word;
   char  *def;
} dictionary[1000] = {
   [0].word = "a",        [0].def = "first letter of the alphabet",
```

```
   [1].word = "aardvark", [1].def = "a burrowing African mammal",
   [2].word = "aback",    [2].def = "to startle"
};
```

or equivalently like this:

```
struct  entry
{
   char  *word;
   char  *def;
} dictionary[1000] = {
   { {.word = "a",        .def = "first letter of the alphabet" },
     {.word = "aardvark", .def = "a burrowing African mammal"} ,
     {.word = "aback",    .def = "to startle"}
};
```

An automatic structure variable can be initialized to another structure of the same type like this:

```
struct date  tomorrow = today;
```

This declares the `date` structure variable `tomorrow` and assigns to it the contents of the (previously declared) `date` structure variable `today`.

A *memberDeclaration* that has the format

```
type  fieldName : n
```

defines a *field* that is *n* bits wide inside the structure, where *n* is an integer value. Fields can be packed from left to right on some machines and from right to left on others. If *fieldName* is omitted, the specified number of bits are reserved, but cannot be referenced. If *fieldName* is omitted and *n* is `0`, the field that follows is aligned on the next storage *unit* boundary, where a *unit* is implementation-defined. The type of field can be `_Bool`, `int`, `signed int`, or `unsigned int`. It is implementation-defined whether an `int` field is treated as `signed` or `unsigned`. The address operator (`&`) cannot be applied to a field, and arrays of fields cannot be defined.

4.3.3 Unions

The general format for declaring a union is as follows:

```
union  name
{
     memberDeclaration
     memberDeclaration
      ...
}  variableList;
```

This defines a union called *name* with members as specified by each *memberDeclaration*. Each member of the union shares overlapping storage space, and the compiler takes care of ensuring that enough space is reserved to contain the largest member of the union.

Variables can be declared at the time that the union is defined, or they can be subsequently declared using the notation

```
union  name  variableList;
```

provided the union was given a name when it was defined.

It is the programmer's responsibility to ensure that the value retrieved from a union is consistent with the last value that was stored inside the union. The *first* member of a union can be initialized by enclosing the initial value, which, in the case of a global union variable, must be a constant expression, inside a pair of curly braces:

```
union  shared
{
    long long  int  l;
    long int      w[2];
} swap = { 0xffffffff };
```

This declares the union variable `swap` and sets the `l` member to hexadecimal `ffffffff`. A different member can be initialized instead by specifying the member name, as in

```
union shared swap2 = {.w[0] = 0x0, .w[1] = 0xffffffff; }
```

An automatic union variable can also be initialized to a union of the same type, as in

```
union shared swap2 = swap;
```

4.3.4 Pointers

The basic format for declaring a pointer variable is as follows:

```
type  *name;
```

The identifier *name* is declared to be of type "pointer to *type*," which can be a basic data type, or a derived data type. For example,

```
int  *ip;
```

declares `ip` to be a pointer to an `int`, and the declaration

```
struct  entry  *ep;
```

declares `ep` to be a pointer to an `entry` structure.

Pointers that point to elements in an array are declared to point to the type of element contained in the array. For example, the previous declaration of `ip` can also be used to declare a pointer into an array of integers.

More advanced forms of pointer declarations are also permitted. For example, the declaration

```
char *tp[100];
```

declares `tp` to be an array of 100 character pointers, and the declaration

```
struct entry (*fnPtr) (int);
```

declares `fnPtr` to be a pointer to a function that returns an `entry` structure and that takes a single `int` argument.

A pointer can be tested to see if it's null by comparing it against a constant expression whose value is `0`. The implementation can choose to internally represent a null pointer with a value other than `0`. However, a comparison between such an internally represented null pointer and a constant value of `0` must prove equal.

The manner in which pointers are converted to integers, and integers are converted to pointers, is machine dependent, as is the size of the integer required to hold a pointer.

The type "pointer to `void`" is the generic pointer type. The language guarantees that a pointer of any type can be assigned to a `void` pointer and back again without changing its value.

Other than this special case, assignment of different pointer types is not permitted, and typically results in a warning message from the compiler if attempted.

4.4 Enumerated Data Types

The general format for declaring enumerated data types is as follows:

```
enum  name { enum_1, enum_2, ... } variableList;
```

The enumerated type *name* is defined with enumeration values *enum_1, enum_2*, ..., each of which is an identifier or an identifier followed by an equal sign and a constant expression. *variableList* is an optional list of variables (with optional initial values) declared to be of type `enum` *name*.

The compiler assigns sequential integers to the enumeration identifiers starting at zero. If an identifier is followed by = and a constant expression, the value of that expression is assigned to the identifier. Subsequent identifiers are assigned values beginning with that constant expression plus 1. Enumeration identifiers are treated as constant integer values by the compiler.

If it is desired to declare variables to be of a previously defined (and named) enumeration type, the construct

```
enum  name  variableList;
```

can be used.

A variable declared to be of a particular enumerated type can only be assigned a value of the same data type, although the compiler might not flag this as an error.

4.5 The `typedef` Statement

The `typedef` statement is used to assign a new name to a basic or derived data type. The `typedef` does not define a new type but simply a new name for an existing type. Therefore, variables declared to be of the newly named type are treated by the compiler exactly as if they were declared to be of the type associated with the new name.

In forming a `typedef` definition, proceed as though a normal variable declaration were being made. Then, place the new type name where the variable name would normally appear. Finally, in front of everything, place the keyword `typedef`.

As an example,

```
typedef struct
        {
             float  x;
             float  y;
        }  Point;
```

associates the name `Point` with a structure containing two floating-point members called `x` and `y`. Variables can subsequently be declared to be of type `Point`, as in

```
Point  origin = { 0.0, 0.0 };
```

4.6 Type Modifiers `const`, `volatile`, and `restrict`

The keyword `const` can be placed before a type declaration to tell the compiler that the value cannot be modified. So the declaration

```
const int x5 = 100;
```

declares `x5` to be a constant integer (that is, it won't be set to anything else during the program's execution). The compiler is *not* required to flag attempts to change the value of a `const` variable.

The `volatile` modifier explicitly tells the compiler that the value changes (usually dynamically). When a `volatile` variable is used in an expression, its value is accessed each place it appears.

To declare `port17` to be of type "volatile pointer to `char`," write

```
volatile char *port17;
```

The `restrict` keyword can be used with pointers. It is a hint to the compiler for optimization (like the `register` keyword for variables). The `restrict` keyword specifies to the compiler that the pointer is the only reference to a particular object; that is, it is not referenced by any other pointer within the same scope. The lines

```
int  * restrict intPtrA;
int  * restrict intPtrB;
```

tell the compiler that for the duration of the scope in which `intPtrA` and `intPtrB` are defined, they will never access the same value. Their use for pointing to integers (in an array, for example) is mutually exclusive.

5.0 Expressions

Variable names, function names, array names, constants, function calls, array references, and structure and union references are all considered expressions. Applying a unary operator (where appropriate) to one of these expressions is also an expression, as is combining two or more of these expressions with a binary or ternary operator. Finally, an expression enclosed within parentheses is also an expression.

An expression of any type other than `void` that identifies a data object is called an `lvalue`. If it can be assigned a value, it is known as a *modifiable lvalue*.

Modifiable lvalue expressions are required in certain places. The expression on the left-hand side of an assignment operator must be a modifiable lvalue. Furthermore, the increment and decrement operators can only be applied to modifiable lvalues, as can the unary address operator `&` (unless it's a function).

5.1 Summary of C Operators

Table A.5 summarizes the various operators in the C language. These operators are listed in order of decreasing precedence. Operators grouped together have the same precedence.

Table A.5 **Summary of C Operators**

Operator	Description	Associativity
`()`	Function call	
`[]`	Array element reference	
`->`	Pointer to structure member reference	Left to right
`.`	Structure member reference	
`-`	Unary minus	
`+`	Unary plus	
`++`	Increment	
`--`	Decrement	
`!`	Logical negation	
`~`	Ones complement	Right to left
`*`	Pointer reference (indirection)	
`&`	Address	
`sizeof`	Size of an object	
(*type*)	Type cast (conversion)	
`*`	Multiplication	
`/`	Division	Left to right
`%`	Modulus	
`+`	Addition	Left to right
`-`	Subtraction	
`<<`	Left shift	Left to right
`>>`	Right shift	
`<`	Less than	
`<=`	Less than or equal to	Left to right
`>`	Greater than	
`=>`	Greater than or equal to	

Table A.5 **Continued**

Operator	Description	Associativity
==	Equality	Left to right
!=	Inequality	
&	Bitwise AND	Left to right
^	Bitwise XOR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
?:	Conditional	Right to left
= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators	Right to left
,	Comma operator	Right to left

As an example of how to use Table A.5, consider the following expression:

```
b | c & d * e
```

The multiplication operator has higher precedence than both the bitwise OR and bitwise AND operators because it appears above both of these in Table A.5. Similarly, the bitwise AND operator has higher precedence than the bitwise OR operator because the former appears above the latter in the table. Therefore, this expression is evaluated as

```
b | ( c & ( d * e ) )
```

Now consider the following expression:

```
b % c * d
```

Because the modulus and multiplication operator appear in the same grouping in Table A.5, they have the same precedence. The associativity listed for these operators is left to right, indicating that the expression is evaluated as

```
( b % c ) * d
```

As another example, the expression

```
++a->b
```

is evaluated as

```
++(a->b)
```

because the `->` operator has higher precedence than the `++` operator.

Finally, because the assignment operators group from right to left, the statement

```
a = b = 0;
```

is evaluated as

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

which has the net result of setting the values of `a` and `b` to `0`. In the case of the expression

```
x[i] + ++i
```

it is not defined whether the compiler evaluates the left side of the plus operator or the right side first. Here, the way that it's done affects the result because the value of `i` might be incremented before `x[i]` is evaluated.

Another case in which the order of evaluation is not defined is in the following expression:

```
x[i] = ++i
```

In this situation, it is not defined whether the value of `i` is incremented before or after its value is used to index into `x`.

The order of evaluation of function arguments is also undefined. Therefore, in the function call

```
f (i, ++i);
```

`i` might be incremented first, thereby causing the same value to be sent as the two arguments to the function.

The C language guarantees that the `&&` and `||` operators are evaluated from left to right. Furthermore, in the case of `&&`, it is guaranteed that the second operand is not evaluated if the first is `0`; and in the case of `||`, it is guaranteed that the second operand is not evaluated if the first is nonzero. This fact is worth bearing in mind when forming expressions such as

```
if ( dataFlag  ||  checkData (myData) )
  ...
```

because, in this case, `checkData` is called only if the value of `dataFlag` is `0`. To take another example, if the array `a` is defined to contain `n` elements, the statement that begins

```
if (index >= 0  &&  index < n  &&  a[index] == 0))
  ...
```

references the element contained in the array only if `index` is a valid subscript in the array.

5.2 Constant Expressions

A constant expression is an expression in which each of the terms is a constant value. Constant expressions are *required* in the following situations:

1. As the value after a case in a `switch` statement
2. For specifying the size of an array that is initialized or globally declared
3. For assigning a value to an enumeration identifier
4. For specifying the bit field size in a structure definition
5. For assigning initial values to static variables
6. For assigning initial values to global variables
7. As the expression following the `#if` in a `#if` preprocessor statement

In the first four cases, the constant expression must consist of integer constants, character constants, enumeration constants, and `sizeof` expressions. The only operators that can be used are the arithmetic operators, the bitwise operators, the relational operators, the conditional expression operator, and the type cast operator. The `sizeof` operator cannot be used on an expression with a variable-length array because the result is evaluated at runtime and is, therefore, not a constant expression.

In the fifth and sixth cases, in addition to the rules cited earlier, the address operator can be implicitly or explicitly used. However, it can only be applied to global or static variables or functions. So, for example, the expression

```
&x + 10
```

is a valid constant expression, provided that `x` is a global or static variable. Furthermore, the expression

```
&a[10] - 5
```

is a valid constant expression if `a` is a global or static array. Finally, because `&a[0]` is equivalent to the expression `a`,

```
a + sizeof (char) * 100
```

is also a valid constant expression.

For the last situation that requires a constant expression (after the `#if`), the rules are the same as for the first four cases, except the `sizeof` operator, enumeration constants, and the type cast operator cannot be used. However, the special `defined` operator is permitted (see Section 9.2.3).

5.3 Arithmetic Operators

Given that

`a, b`	are expressions of any basic data type except `void`;
`i, j`	are expressions of any integer data type;

then the expression

`-a`	negates the value of `a`;
`+a`	gives the value of `a`;
`a + b`	adds `a` with `b`;

`a - b`	subtracts `b` from `a`;
`a * b`	multiplies `a` by `b`;
`a / b`	divides `a` by `b`;
`i % j`	gives the remainder of `i` divided by `j`.

In each expression, the usual arithmetic conversions are performed on the operands (see Section 5.17). If `a` is unsigned, `-a` is calculated by first applying integral promotion to it, subtracting it from the largest value of the promoted type, and adding 1 to the result.

If two integral values are divided, the result is truncated. If either operand is negative, the direction of the truncation is not defined (that is, –3 / 2 might produce –1 on some machines and –2 on others); otherwise, truncation is always toward zero (3 / 2 always produces 1). See Section 5.15 for a summary of arithmetic operations with pointers.

5.4 Logical Operators

Given that

`a, b`	are expressions of any basic data type except `void`, or are both pointers;

then the expression

`a && b`	has the value `1` if both `a` and `b` are nonzero, and `0` otherwise (and `b` is evaluated only if `a` is nonzero);
`a || b`	has the value `1` if either `a` or `b` is nonzero, and `0` otherwise (and `b` is evaluated only if `a` is zero);
`! a`	has the value `1` if `a` is zero, and `0` otherwise.

The usual arithmetic conversions are applied to `a` and `b` (see Section 5.17). The type of the result in all cases is `int`.

5.5 Relational Operators

Given that

`a, b`	are expressions of any basic data type except `void`, or are both pointers;

then the expression

`a < b`	has the value `1` if `a` is less than `b`, and `0` otherwise;
`a <= b`	has the value `1` if `a` is less than or equal to `b`, and `0` otherwise;
`a > b`	has the value `1` if `a` is greater than `b`, and `0` otherwise;
`a >= b`	has the value `1` if `a` is greater than or equal to `b`, and `0` otherwise;
`a == b`	has the value `1` if `a` is equal to `b`, and `0` otherwise;
`a != b`	has the value `1` if `a` is not equal to `b`, and `0` otherwise.

The usual arithmetic conversions are performed on `a` and `b` (see Section 5.17). The first four relational tests are only meaningful for pointers if they both point into the same array or to members of the same structure or union. The type of the result in each case is `int`.

5.6 Bitwise Operators

Given that

`i, j, n`	are expressions of any integer data type;

then the expression

`i & j`	performs a bitwise AND of `i` and `j`;
`i | j`	performs a bitwise OR of `i` and `j`;
`i ^ j`	performs a bitwise XOR of `i` and `j`;
`~i`	takes the ones complement of `i`;
`i << n`	shifts `i` to the left `n` bits;
`i >> n`	shifts `i` to the right `n` bits.

The usual arithmetic conversions are performed on the operands, except with `<<` and `>>`, in which case just integral promotion is performed on each operand (see Section 5.17). If the shift count is negative or is greater than or equal to the number of bits contained in the object being shifted, the result of the shift is undefined. On some machines, a right shift is arithmetic (sign fill) and on others logical (zero fill). The type of the result of a shift operation is that of the promoted left operand.

5.7 Increment and Decrement Operators

Given that

`lv`	is a modifiable `lvalue` expression, whose type is not qualified as `const`;

then the expression

`++lv`	increments `lv` and then uses its value as the value of the expression;
`lv++`	uses `lv` as the value of the expression and then increments `lv`;
`--lv`	decrements `lv` and then uses its value as the value of the expression;
`lv--`	uses `lv` as the value of the expression and then decrements `lv`.

Section 5.15 describes these operations on pointers.

5.8 Assignment Operators

Given that

`lv`	is a modifiable `lvalue` expression, whose type is not qualified as `const`;
`op`	is any operator that can be used as an assignment operator (see Table A.5);
`a`	is an expression;

then the expression

`lv = a`	stores the value of `a` into `lv`;
`lv` *`op`*`= a`	applies *`op`* to `lv` and `a`, storing the result in `lv`.

In the first expression, if `a` is one of the basic data types (except `void`), it is converted to match the type of `lv`. If `lv` is a pointer, `a` must be a pointer to the same type as `lv`, a `void` pointer, or the *null* pointer.

If `lv` is a `void` pointer, `a` can be of any pointer type. The second expression is treated as if it were written `lv = lv` *`op`* `(a)`, except `lv` is only evaluated once (consider `x[i++] += 10`).

5.9 Conditional Operators

Given that

`a, b, c`	are expressions;

then the expression

`a ? b : c`	has as its value `b` if `a` is nonzero, and `c` otherwise; only expression `b` or `c` is evaluated.

Expressions `b` and `c` must be of the same data type. If they are not, but are both arithmetic data types, the usual arithmetic conversions are applied to make their types the same. If one is a pointer and the other is zero, the latter is taken as a null pointer of the same type as the former. If one is a pointer to `void` and the other is a pointer to another type, the latter is converted to a pointer to `void`, and that is the resulting type.

5.10 Type Cast Operator

Given that

`type`	is the name of a basic data type, an enumerated data type (preceded by the keyword `enum`), a `typedef`-defined type, or is a derived data type;
`a`	is an expression;

then the expression

`(` *`type`* `)`	converts `a` to the specified type.

5.11 `sizeof` Operator

Given that

type	is as described previously;
a	is an expression;

then the expression

sizeof (*type*)	has as its value the number of bytes needed to contain a value of the specified type;
sizeof a	has as its value the number of bytes required to hold the result of the evaluation of a.

If *type* is char, the result is defined to be 1. If a is the name of an array that has been dimensioned (either explicitly or implicitly through initialization) and is not a formal parameter or undimensioned extern array, sizeof a gives the number of bytes required to store the elements in a.

The type of integer produced by the sizeof operator is size_t, which is defined in the standard header file <stddef.h>.

If a is a variable-length array, the sizeof operator is evaluated at runtime; otherwise a is evaluated at compile time and the result can be used in constant expressions (see Section 5.2).

5.12 Comma Operator

Given that

a, b	are expressions;

then the expression

a, b	causes a to be evaluated and then b to be evaluated; the type and value of the expression is that of b.

5.13 Basic Operations with Arrays

Given that

a	is declared as an array of n elements;
i	is an expression of any integer data type;
v	is an expression;

then the expression

a[0]	references the first element of a;
a[n - 1]	references the last element of a;
a[i]	references element number i of a;
a[i] = v	stores the value of v into a[i].

In each case, the type of the result is the type of the elements contained in `a`. See Section 5.15 for a summary of operations with pointers and arrays.

5.14 Basic Operations with Structures[1]

Given that

`x`	is a modifiable lvalue expression of type `struct s`;
`y`	is an expression of type `struct s`;
`m`	is the name of one of the members of the structure `s`;
`v`	is an expression;

then the expression

`x`	references the entire structure and is of type `struct s`;
`y.m`	references the member `m` of the structure `y` and is of the type declared for the member `m`;
`x.m = v`	assigns `v` to the member `m` of `x` and is of the type declared for the member `m`;
`x = y`	assigns `y` to `x` and is of type `struct s`;
`f (y)`	calls the function `f`, passing contents of the structure `y` as the argument; inside `f`, the formal parameter must be declared to be of type `struct s`;
`return y;`	returns the structure `y;` the return type declared for the function must be `struct s`.

5.15 Basic Operations with Pointers

Given that

`x`	is an `lvalue` expression of type `t`;
`pt`	is a modifiable lvalue expression of type "pointer to `t`";
`v`	is an expression;

then the expression

`&x`	produces a pointer to `x` and has type "pointer to `t`";
`pt = &x`	sets `pt` pointing to `x` and has type "pointer to `t`";
`pt = 0`	assigns the null pointer to `pt`;
`pt == 0`	tests to see if `pt` is null;
`*pt`	references the value pointed to by `pt` and has type `t`;
`*pt = v`	stores the value of `v` into the location pointed to by `pt` and has type `t`.

1. Also applies to unions.

Pointers to Arrays

Given that

`a`	is an array of elements of type *t*;
`pa1`	is a modifiable lvalue expression of type "pointer to *t*" that points to an element in `a`;
`pa2`	is an `lvalue` expression of type "pointer to *t*" that points to an element in `a`, or to one past the last element in `a`;
`v`	is an expression;
`n`	is an integral expression;

then the expression

`a`, `&a`, `&a[0]`	produces a pointer to the first element;
`&a[n]`	produces a pointer to element number `n` of `a` and has type "pointer to *t*";
`*pa1`	references the element of `a` that `pa1` points to and has type *t*;
`*pa1 = v`	stores the value of `v` in the element pointed to by `pa1`, and has type *t*;
`++pa1`	sets `pa1` pointing to the next element of `a`, no matter what type of elements are contained in `a` and has type "pointer to *t*";
`--pa1`	sets `pa1` pointing to the previous element of `a`, no matter what type of elements are contained in `a`, and has type "pointer to *t*";
`*++pa1`	increments `pa1` and then references the value in `a` that `pa1` points to, and has type *t*;
`*pa1++`	references the value in `a` that `pa1` points to before incrementing `pa1` and has type *t*;
`pa1 + n`	produces a pointer that points `n` elements further into `a` than `pa1` and has type "pointer to *t*";
`pa1 - n`	produces a pointer to `a` that points `n` elements previous to that pointed to by `pa1` and has type "pointer to *t*";
`*(pa1 + n) = v`	stores the value of `v` in the element pointed to by `pa1 + n` and has type *t*;
`pa1 < pa2`	tests if `pa1` is pointing to an earlier element in `a` than is `pa2` and has type `int` (any relational operators can be used to compare two pointers);
`pa2 - pa1`	produces the number of elements in `a` contained between the pointers `pa2` and `pa1` (assuming that `pa2` points to an element further in `a` than `pa1`) and has integer type;

`a + n`	produces a pointer to element number `n` of `a`, has type "pointer to *t*," and is in all ways equivalent to the expression `&a[n]`;
`*(a + n)`	references element number `n` of `a`, has type *t*, and is in all ways equivalent to the expression `a[n]`.

The actual type of the integer produced by subtracting two pointers is specified by `ptrdiff_t`, which is defined in the standard header file `<stddef.h>`.

Pointers to Structures[2]

Given that

`x`	is an `lvalue` expression of type `struct` *s*;
`ps`	is a modifiable lvalue expression of type "pointer to `struct` *s*";
`m`	is the name of a member of the structure *s* and is of type *t*;
`v`	is an expression;

then the expression

`&x`	produces a pointer to `x` and is of type "pointer to `struct` *s*";
`ps = &x`	sets `ps` pointing to `x` and is of type "pointer to `struct` *s*";
`ps->m`	references member `m` of the structure pointed to by `ps` and is of type *t*;
`(*ps).m`	also references this member and is in all ways equivalent to the expression `ps->m`;
`ps->m = v`	stores the value of `v` into the member `m` of the structure pointed to by `ps` and is of type *t*.

5.16 Compound Literals

A compound literal is a type name enclosed in parentheses followed by an initialization list. It creates an unnamed value of the specified type, which has scope limited to the block in which it is created, or global scope if defined outside of any block. In the latter case, the initializers must all be constant expressions.

As an example,

```
(struct  point) {.x = 0, .y = 0}
```

is an expression that produces a structure of type `struct point` with the specified initial values. This can be assigned to another `struct point` structure, as in

```
origin = (struct  point) {.x = 0, .y = 0};
```

or passed to a function expecting an argument of `struct point`, as in

```
moveToPoint ((struct  point) {.x = 0, .y = 0});
```

2. Also applies to unions.

Types other than structures can be defined as well, for example, if `intPtr` is of type `int *`, the statement

```
intPtr = (int [100]) {[0] = 1, [50] = 50, [99] = 99 };
```

(which can appear anywhere in the program) sets `intptr` pointing to an array of 100 integers, whose three elements are initialized as specified.

If the size of the array is not specified, it is determined by the initializer list.

5.17 Conversion of Basic Data Types

The C language converts operands in arithmetic expressions in a predefined order, known as the *usual arithmetic conversions*.

Step 1: If either operand is of type `long double`, the other is converted to `long double`, and that is the type of the result.

Step 2: If either operand is of type `double`, the other is converted to `double`, and that is the type of the result.

Step 3: If either operand is of type `float`, the other is converted to `float`, and that is the type of the result.

Step 4: If either operand is of type `_Bool`, `char`, `short int`, `int` bit field, or of an enumerated data type, it is converted to `int`, if an `int` can fully represent its range of values; otherwise, it is converted to `unsigned int`. If both operands are of the same type, that is the type of the result.

Step 5: If both operands are signed or both are unsigned, the smaller integer type is converted to the larger integer type, and that is the type of the result.

Step 6: If the unsigned operand is equal in size or larger than the signed operand, then the signed operand is converted to the type of the unsigned operand, and that is the type of the result.

Step 7: If the signed operand can represent all of the values in the unsigned operand, the latter is converted to the type of the former, and that is the type of the result.

Step 8: If this step is reached, both operands are converted to the unsigned type corresponding to the type of the signed type.

Step 4 is known more formally as *integral promotion*.

Conversion of operands is well behaved in most situations, although the following points should be noted:

1. Conversion of a `char` to an `int` might involve sign extension on some machines, unless the `char` is declared as `unsigned`.

2. Conversion of a signed integer to a longer integer results in extension of the sign to the left; conversion of an unsigned integer to a longer integer results in zero fill to the left.
3. Conversion of any value to a `_Bool` results in `0` if the value is zero and `1` otherwise.
4. Conversion of a longer integer to a shorter one results in truncation of the integer on the left.
5. Conversion of a floating-point value to an integer results in truncation of the decimal portion of the value. If the integer is not large enough to contain the converted floating-point value, the result is not defined, as is the result of converting a negative floating-point value to an unsigned integer.
6. Conversion of a longer floating-point value to a shorter one might or might not result in rounding before the truncation occurs.

6.0 Storage Classes and Scope

The term *storage class* refers to the manner in which memory is allocated by the compiler in the case of variables and to the scope of a particular function definition. Storage classes are `auto`, `static`, `extern`, and `register`. A storage class can be omitted in a declaration and a default storage class is assigned, as discussed later in this chapter.

The term *scope* refers to the extent of the meaning of a particular identifier within a program. An identifier defined outside any function or statement block (herein referred to as a *BLOCK*) can be referenced anywhere subsequent in the file. Identifiers defined within a BLOCK are local to that BLOCK and can locally redefine an identifier defined outside it. Label names are known throughout the BLOCK, as are formal parameter names. Labels, structure and structure member names, union and union member names, and enumerated type names do not have to be distinct from each other or from variable or function names. However, enumeration identifiers *do* have to be distinct from variable names and from other enumeration identifiers defined within the same scope.

6.1 Functions

If a storage class is specified when a function is defined, it must be either `static` or `extern`. Functions that are declared as `static` can only be referenced from within the same file that contains the function. Functions that are specified as `extern` (or that have no class specified) can be called by functions from other files.

6.2 Variables

Table A.6 summarizes the various storage classes that can be used in declaring variables as well as their scope and methods of initialization.

Table A.6 **Variables: Summary of Storage Classes, Scope, and Initialization**

If Storage Class is	And Variable is declared	Then it can be referenced	And can be initialized with	Comments
`static`	Outside any BLOCK Inside a BLOCK	Anywhere within the file Within the BLOCK	Constant expression only	Variables are initialized only once at the start of program execution; values are retained through BLOCKs; default value is 0
`extern`	Outside any BLOCK Inside a BLOCK	Anywhere within the file Within the BLOCK	Constant expression only	Variable must be declared in at least one place without the `extern` keyword, or in one place using the keyword `extern` and assigned an initial value
`auto`	Inside a BLOCK	Within the BLOCK	Any valid expression	Variable is initialized each time the BLOCK is entered; no default value
`register`	Inside a BLOCK	Within the BLOCK	Any valid expression	Assignment to register not guaranteed; varying restrictions on types of variables that can be declared; cannot take the address of a register variable; initialized each time BLOCK is entered; no default value

Table A.6 **Continued**

If Storage Class is	And Variable is declared	Then it can be referenced	And can be initialized with	Comments
`omitted`	Outside any BLOCK	Anywhere within the file or by other files that contain appropriate declarations	Constant expressions only	This declaration can appear in only one place; variable is initialized at the start of program execution; default value is 0
	Inside a BLOCK	(See `auto`)	(See `auto`)	defaults to `auto`

7.0 Functions

This section summarizes the syntax and operation of functions.

7.1 Function Definition

The general format for declaring a function definition is as follows:

```
returnType   name ( type1 param1, type2 param2, ... )
{
     variableDeclarations

     programStatement
     programStatement
     ...
     return expression;
}
```

The function called *name* is defined, which returns a value of type *returnType* and has formal parameters *param1*, *param2*, . . . *param1* is declared to be of type *type1*, *param2* is declared to be of type *type2*, and so on.

Local variables are typically declared at the beginning of the function, but that's not required. They can be declared anywhere, in which case their access is limited to statements appearing after their declaration in the function.

If the function does not return a value, *returnType* is specified as `void`.

If just `void` is specified inside the parentheses, the function takes no arguments. If . . . is used as the last (or only) parameter in the list, the function takes a variable number of arguments, as in

```
int  printf (char *format, ...)
{
   ...
}
```

Declarations for single-dimensional array arguments do not have to specify the number of elements in the array. For multidimensional arrays, the size of each dimension except the first must be specified.

See Section 8.9 for a discussion of the `return` statement.

The keyword `inline` can be placed in front of a function definition as a hint to the compiler. Some compilers replace the function call with the actual code for the function itself, thus providing for faster execution. An example is

```
inline int min (int a, int b)
{
    return ( a < b ? a : b);
}
```

7.2 Function Call

The general format for declaring a function call is as follows:

name (*arg1*, *arg2*, ...)

The function called *name* is called and the values *arg1*, *arg2*, ... are passed as arguments to the function. If the function takes no arguments, just the open and closed parentheses are specified (as in `initialize ()`).

If you are calling a function that is defined after the call, or in another file, you should include a *prototype declaration* for the function, which has the following general format:

returnType name (type1 param1, type2 param2, ...);

This tells the compiler the function's return type, the number of arguments it takes, and the type of each argument. As an example, the line

```
long double  power (double x, int n);
```

declares `power` to be a function that returns a `long double` and that takes two arguments, the first a `double` and the second an `int`. The argument names inside the parentheses are actually dummy names and can be omitted if desired, so

```
long double  power (double, int);
```

works just as well.

If the compiler has previously encountered the function definition or a prototype declaration for the function, the type of each argument is automatically converted (where possible) to match the type expected by the function when the function is called.

If neither the function's definition nor a prototype declaration has been encountered, the compiler assumes the function returns a value of type `int`, automatically converts all `float` arguments to type `double`, and performs integral promotion on any integer arguments as outlined in Section 5.17. Other function arguments are passed without conversion.

Functions that take a variable number of arguments must be declared as such. Otherwise, the compiler is at liberty to assume the function takes a fixed number of arguments based upon the number actually used in the call.

A function whose return type is declared as `void` causes the compiler to flag any calls to that function that try to make use of a returned value.

All arguments to a function are passed by value; therefore, their values cannot be changed by the function. If a pointer is passed to a function, the function *can* change values referenced by the pointer, but it still cannot change the value of the pointer variable itself.

7.3 Function Pointers

A function name, without a following set of parentheses, produces a pointer to that function. The address operator can also be applied to a function name to produce a pointer to it.

If `fp` is a pointer to a function, the corresponding function can be called either by writing

```
fp ()
```

or

```
(*fp) ()
```

If the function takes arguments, they can be listed inside the parentheses.

8.0 Statements

A program statement is any valid expression (usually an assignment or function call) that is immediately followed by a semicolon, or it is one of the special statements described in the following sections. A *label* can optionally precede any statement, and consists of an identifier followed immediately by a colon (see Section 8.6).

8.1 Compound Statements

Program statements that are contained within a pair of braces are known collectively as a *compound* statement or *block* and can appear anywhere in the program that a single statement is permitted. A block can have its own set of variable declarations, which override any similarly named variables defined outside the block. The scope of such local variables is the block in which they are defined.

8.2 The `break` Statement

The general format for declaring a `break` statement is as follows:

```
break;
```

Execution of a `break` statement from within a `for`, `while`, `do`, or `switch` statement causes execution of that statement to be immediately terminated. Execution continues with the statement that immediately follows the loop or switch.

8.3 The `continue` Statement

The general format for declaring the `continue` statement is as follows:

```
continue;
```

Execution of the `continue` statement from within a loop causes any statements that follow the `continue` in the loop to be skipped. Execution of the loop otherwise continues as normal.

8.4 The `do` Statement

The general format for declaring the `do` statement is as follows:

```
do
      programStatement
while ( expression );
```

`programStatement` is executed as long as *`expression`* evaluates as nonzero. Note that, because *`expression`* is evaluated each time *after* the execution of *`programStatement`*, it is guaranteed that *`programStatement`* will be executed at least once.

8.5 The `for` Statement

The general format for declaring the `for` statement is as follows:

```
for ( expression_1;  expression_2; expression_3 )
     programStatement
```

`expression_1` is evaluated once when execution of the loop begins. Next, *`expression_2`* is evaluated. If its value is nonzero, *`programStatement`* is executed and then *`expression_3`* is evaluated. Execution of *`programStatement`* and the subsequent evaluation of *`expression_3`* continues as long as the value of *`expression_2`* is nonzero. Note that, because *`expression_2`* is evaluated each time before *`programStatement`* is executed, *`programStatement`* might never be executed if the value of *`expression_2`* is `0` when the loop is first entered.

Variables local to the `for` loop can be declared in *`expression_1`*. The scope of such variables is the scope of the `for` loop. For example,

```
for ( int i = 0; i < 100; ++i)
    ...
```

declares the integer variable `i` and sets its initial value to `0` when the loop begins. The variable can be accessed by any statements inside the loop, but is not accessible after the loop is terminated.

8.6 The goto Statement

The general format for declaring the goto statement is as follows:

```
goto identifier;
```

Execution of the goto causes control to be sent directly to the statement labeled *identifier*. The labeled statement must be located in the same function as the goto.

8.7 The if Statement

One general format for declaring an if statement is as follows:

```
if ( expression )
    programStatement
```

If the result of evaluating *expression* is nonzero, *programStatement* is executed; otherwise, it is skipped.

Another general format for declaring an if statement is as follows:

```
if ( expression )
    programStatement_1
else
    programStatement_2
```

If the value of *expression* is nonzero, the *programStatement_1* is executed; otherwise, *programStatement_2* is executed. If *programStatement_2* is another if statement, an if-else if chain is affected:

```
if ( expression_1 )
    programStatement_1
else if ( expression_2 )
    programStatement_2
    ...
else
    programStatement_n
```

An else clause is always associated with the last if statement that does not contain an else. Braces can be used to change this association if necessary.

8.8 The null Statement

The general format for declaring the null statement is as follows:

```
;
```

Execution of a null statement has no effect and is used primarily to satisfy the requirement of a program statement in a for, do, or while loop. For example, in the following statement, which copies a character string pointed to by from to one pointed to by to:

```
while ( *to++ = *from++ )
    ;
```

the `null` statement is used to satisfy the requirement that a program statement appear after the looping expression of the `while`.

8.9 The `return` Statement

One general format for declaring the `return` statement is as follows:

```
return;
```

Execution of the `return` statement causes program execution to be immediately returned to the calling function. This format can only be used to return from a function that does not return a value.

If execution proceeds to the end of a function and a `return` statement is not encountered, it returns as if a `return` statement of this form had been executed. Therefore, in such a case, no value is returned.

A second general format for declaring the `return` statement is as follows:

```
return  expression;
```

The value of *`expression`* is returned to the calling function. If the type of *`expression`* does not agree with the return type declared in the function declaration, its value is automatically converted to the declared type before it is returned.

8.10 The `switch` Statement

The general format for declaring a `switch` statement is as follows:

```
switch ( expression )
{
    case constant_1:
        programStatement
        programStatement
          ...
        break;
    case constant_2:
        programStatement
        programStatement
          ...
        break;
    ...
    case constant_n:
        programStatement
        programStatement
          ...
        break;
    default:
        programStatement
        programStatement
          ...
        break;
}
```

expression is evaluated and compared against the constant expression values *constant_1*, *constant_2*, ..., *constant_n*. If the value of *expression* matches one of these case values, the program statements that immediately follow are executed. If no case value matches the value of *expression*, the default case, if included, is executed. If the default case is not included, no statements contained in the switch are executed.

The result of the evaluation of *expression* must be of integral type and no two cases can have the same value. Omitting the `break` statement from a particular case causes execution to continue into the next case.

8.11 The `while` Statement

The general format for declaring the `while` statement is as follows:

```
while ( expression )
    programStatement
```

programStatement is executed as long as the value of *expression* is nonzero. Note that, because *expression* is evaluated each time *before* the execution of *programStatement*, *programStatement* might never be executed.

9.0 The Preprocessor

The preprocessor analyzes the source file before the compiler properly sees the code. The preprocessor does the following:

1. Replaces trigraph sequences (See Section 9.1) by their equivalents
2. Joins any lines that end with a backslash character (\) together into a single line
3. Divides the program into a stream of tokens
4. Removes comments, replacing them with a single space
5. Processes preprocessor directives (see Section 9.2) and expands macros

9.1 Trigraph Sequences

To handle non-ASCII character sets, the three-character sequences (called *trigraphs*) listed in Table A.7 are recognized and treated specially wherever they occur inside a program (as well as inside character strings):

Table A.7 **Trigraph Sequences**

Trigraph	Meaning
??=	#
??(	[
??)	]
??<	{

Table A.7 **Trigraph Sequences**

Trigraph	Meaning
??>	}
??/	\
??'	^
??!	|
??-	~

9.2 Preprocessor Directives

All preprocessor directives begin with the character #, which must be the first nonwhite-space character on the line. The # can be optionally followed by one or more space or tab characters.

9.2.1 The `#define` Directive

The general format for declaring the `#define` directive is as follows:

```
#define name  text
```

This defines the identifier *name* to the preprocessor and associates with it whatever *text* appears after the first blank space after *name* to the end of the line. Subsequent use of *name* in the program causes *text* to be substituted directly into the program at that point.

Another general format for declaring the `#define` directive is as follows:

```
#define name(param_1, param_2, ..., param_n)  text
```

The macro *name* is defined to take arguments as specified by *param_1*, *param_2*, ..., *param_n*, each of which is an identifier. Subsequent use of *name* in the program with an argument list causes *text* to be substituted directly into the program at that point, with the arguments of the macro call replacing all occurrences of the corresponding parameters inside *text*.

If the macro takes a variable number of arguments, three dots are used at the end of the argument list. The remaining arguments in the list are collectively referenced in the macro definition by the special identifier `__VA_ARGS__`. As an example, the following defines a macro called `myPrintf` to take a leading format string followed by a variable number of arguments:

```
#define myPrintf(...)   printf ("DEBUG: " __VA_ARGS__);
```

Legitimate macro uses includes

```
myPrintf ("Hello world!\n");
```

as well as

```
myPrintf ("i = %i, j = %i\n", i, j);
```

If a definition requires more than one line, each line to be continued must be ended with a backslash character. After a name has been defined, it can be used subsequently anywhere in the file.

The # operator is permitted in `#define` directives that take arguments. It is followed by the name of an argument to the macro. The preprocessor puts double quotation marks around the actual value passed to the macro when it's invoked. That is, it turns it into a character string. For example, the definition

```
#define  printint(x)  printf (# x " = %d\n", x)
```

with the call

```
printint (count);
```

is expanded by the preprocessor into

```
printf ("count" " = %i\n", count);
```

or, equivalently,

```
printf ("count = %i\n", count);
```

The preprocessor puts a \ character in front of any " or \ characters when performing this *stringizing* operation. So, with the definition

```
#define  str(x)  # x
```

the call

```
str (The string "\t" contains a tab)
```

expands to

```
"The string \"\\t\" contains a tab"
```

The ## operator is also allowed in `#define` directives that take arguments. It is preceded (or followed) by the name of an argument to the macro. The preprocessor takes the value that is passed when the macro is invoked and creates a single token from the argument to the macro and the token that follows (or precedes) it. For example, the macro definition

```
#define printx(n)  printf ("%i\n", x ## n );
```

with the call

```
printx (5)
```

produces

```
printf ("%i\n", x5);
```

The definition

```
#define printx(n) printf ("x" # n " = %i\n", x ## n );
```

with the call

```
printx(10)
```

produces

```
printf ("x10 = %i\n", x10);
```

after substitution and concatenation of the character strings.

Spaces are not required around the # and ## operators.

9.2.2 The #error Directive

The general format for declaring the #error directive is as follows:

```
#error text
   ...
```

The specified *text* is written as an error message by the preprocessor.

9.2.3 The #if Directive

One general format for declaring the #if directive is as follows:

```
#if constant_expression
   ...
#endif
```

The value of *constant_expression* is evaluated. If the result is nonzero, all program lines up until the #endif directive are processed; otherwise, they are automatically skipped and are not processed by the preprocessor or by the compiler.

Another general format for declaring the #if directive is as follows:

```
#if constant_expression_1
   ...
#elif constant_expression_2
   ...
#elif constant_expression_n
   ...
#else
   ...
#endif
```

If *constant_expression_1* is nonzero, all program lines up until the #elif are processed, and the remaining lines up to the #endif are skipped. Otherwise, if *constant_expression_2* is nonzero, all program lines up until the next #elif are processed, and the remaining lines up to the #endif are skipped. If none of the constant expressions evaluates to nonzero, the lines after the #else (if included) are processed.

The special operator defined can be used as part of the constant expression, so

```
#if defined (DEBUG)
   ...
#endif
```

causes the code between the `#if` and `#endif` to be processed if the identifier `DEBUG` has been previously defined (see Section 9.2.4). The parentheses are not necessary around the identifier, so

```
#if defined DEBUG
```

works just as well.

9.2.4 The `#ifdef` Directive

The general format for declaring the `#ifdef` directive is as follows:

```
#ifdef identifier
  ...
#endif
```

If the value of *`identifier`* has been previously defined (either through a `#define` or with the `-D` command-line option when the program was compiled), all program lines up until the `#endif` are processed; otherwise, they are skipped. As with the `#if` directive, `#elif` and `#else` directives can be used with a `#ifdef` directive.

9.2.5 The `#ifndef` Directive

The general format for declaring the `#ifndef` directive is as follows:

```
#ifndef identifier
  ...
#endif
```

If the value of *`identifier`* has not been previously defined, all program lines up until the `#endif` are processed; otherwise, they are skipped. As with the `#if` directive, `#elif` and `#else` directives can be used with a `#ifndef` directive.

9.2.6 The `#include` Directive

One general format for declaring the `#include` directive is as follows:

```
#include "fileName"
```

The preprocessor searches an implementation-defined directory or directories first for the file *`fileName`*. Typically, the same directory that contains the source file is searched first. If the file is not found there, a sequence of implementation-defined standard places is searched. After it is found, the contents of the file is included in the program at the precise point that the `#include` directive appears. Preprocessor directives contained within the included file are analyzed, and, therefore, an included file can itself contain other `#include` directives.

Another general format for declaring the `#include` directive is as follows:

```
#include <fileName>
```

The preprocessor searches for the specified file only in the standard places. The action taken after the file is found is otherwise identical to that described previously.

In either format, a previously defined name can be supplied and expansion occurs. So the following sequence works:

```
#define DATABASE_DEFS    </usr/data/database.h>
    …
#include DATABASE_DEFS
```

9.2.7 The `#line` Directive

The general format for declaring the `#line` directive is as follows:

```
#line  constant  "fileName"
```

This directive causes the compiler to treat subsequent lines in the program as if the name of the source file were *`fileName`*, and as if the line number of all subsequent lines began at *`constant`*. If *`fileName`* is not specified, the filename specified by the last `#line` directive or the name of the source file (if no filename was previously specified) is used.

The `#line` directive is primarily used to control the filename and line number that are displayed whenever an error message is issued by the compiler.

9.2.8 The `#pragma` Directive

The general format for declaring the `#pragma` directive is as follows:

```
#pragma text
```

This causes the preprocessor to perform some implementation-defined action. For example,

```
#pragma loop_opt(on)
```

might cause special loop optimization to be performed on a particular compiler. If this pragma is encountered by a compiler that doesn't recognize the `loop_opt` pragma, it is ignored.

The special keyword `STDC` is used after the `#pragma` for special meaning. Current supported "switches" that can follow a `#pragma STDC` are `FP_CONTRACT`, `FENV_ACCESS`, and `CX_LIMITED_RANGE`.

9.2.9 The `#undef` Directive

The general format for declaring the `#undef` directive is as follows:

```
#undef  identifier
```

The specified *`identifier`* becomes undefined to the preprocessor. Subsequent `#ifdef` or `#ifndef` directives behave as if the identifier were never defined.

9.2.10 The # Directive

This is a null directive and is ignored by the preprocessor.

9.3 Predefined Identifiers

The identifiers listed in Table A.8 are defined by the preprocessor.

Table A.8 **Predefined Preprocesor Identifiers**

Identifier	Meaning
`__LINE__`	Current line number being compiled
`__FILE__`	Name of the current source file being compiled
`__DATE__`	Date the file is being compiled, in the format "*mm dd yyyy*"
`__TIME__`	Time the file is being compiled, in the format "*hh:mm:ss*"
`__STDC__`	Defined as `1` if the compiler conforms to the ANSI standard, `0` if not
`__STDC_HOSTED__`	Defined as `1` if the implementation is hosted, `0` if not
`__STDC_VERSION__`	Defined as 199901L