The GNU C Reference Manual

Trevis J. Rothwell

This is the GNU C reference manual.

Table of Contents

Preface

This is a reference manual for the GNU C programming language: the C programming language as implemented by the GNU C compiler.

GCC supports several variants of C; this manual ultimately aims to explicitly document three of them:

- The original 1989 ANSI C standard, commonly known as "C89"
- The revised 1999 ISO C standard, commonly known as "C99"
- The current state of GNU extensions to standard C

By default, GCC will compile code as C89 plus GNU-specific extensions. Much of C99 is supported; once full support is available, the default compilation dialect will be C99 plus GNU-specific extensions. (Note that some of the GNU extensions to C89 ended up, sometimes slightly modified, as standard language features in C99.)

Presently, this manual describes C89 only. Descriptions of C99 and GNU extensions will be included in future releases, labeled as such alongside the original C89 material. While most users of GCC are free to use the latest and greatest additions to the compiler, some users must continue to use older versions of GCC. (For example, avionics programmers cannot switch to newer compilers on a whim, and indeed often use the same compiler for years until a new version is approved for use.) For this reason, we feel that a clear distinction between C dialects will be useful to our readers.

The C language includes a set of preprocessor directives, which are used for things such as macro text replacement, conditional compilation, and file inclusion. Although normally described in a C language manual, the GNU C preprocessor has been thoroughly documented in *The C Preprocessor*, a separate manual which covers preprocessing for C, C++, and Objective C programs, so it is not included here.

A number of people have contributed to this manual. Most of the text was written by Trevis Rothwell. Other contributors, who have offered editing, proofreading, ideas, and/or help with typesetting or administrative details, include: Karl Berry, Robert Chassell, Lisa Goldstein, Robert Hansen, Jean-Christophe Helary, Joe Humphries, J. Wren Hunt, Adam Johansen, Steve Morningthunder, Richard Stallman, J. Otto Tennant, Ole Tetlie, and T.F. Torrey.

Please send bug reports and suggestions to `gnu-c-manual@gnu.org`.

1 Lexical Elements

This chapter describes the lexical elements that make up C source code after preprocessing. These basic elements are called *tokens*, and there are several distinct types of tokens: keywords, identifiers, constants, operators, and separators. White space, sometimes required to separate tokens, is also described in this chapter.

1.1 Identifiers

Identifiers are strings of characters used for naming variables, functions, new data types, and preprocessor macros. The characters can be letters, decimal digits, the underscore character '`_`', or the dollar sign character '`$`'.

The first character of an identifier cannot be a digit.

Lowercase letters and uppercase letters are distinct, so `squid` and `SQUID` are two different identifiers.

1.2 Keywords

Keywords are special identifiers, reserved for use by the programming language itself. You cannot use them for any other purpose.

Here is a list of keywords recognized by ANSI C89:

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

1.3 Constants

A constant is a literal numeric or character value, such as 5 or 'm'. All constants are of a particular data type; you can use type casting to explicitly specify the type of a constant, or let the compiler use the default type based on the value of the constant.

1.3.1 Integer Constants

An integer constant is a sequence of digits.

If the sequence of digits is preceded by `0x` or `0X` (zero x or zero X), then the constant is considered to be hexadecimal (base 16). Hexadecimal values may use the digits from 0 to 9, as well as the letters `a` to `f` and `A` to `F`.

```
0x2f
0x88
0xAB43
0xAbCd
0x1
```

If the first digit is 0 (zero), and the next character is not `x` or `X`, then the constant is considered to be octal (base 8). Octal values may only use the digits from 0 to 7; 8 and 9 are not allowed.

```
057
012
03
0841
```

In all other cases, the sequence of digits is assumed to be decimal (base 10). Decimal values may use the digits from 0 to 9.

```
459
23901
8
12
```

There are various integer data types, for short integers, long integers, signed integers, and unsigned integers. You can force an integer constant to be of a long and/or unsigned integer type by appending a sequence of one or more letters to the end of the constant:

`u`
`U` Unsigned integer type.

`l`
`L` Long integer type.

For example, `45U` is an `unsigned int` constant. You can also combine letters: `45UL` is an `unsigned long int` constant. (The letters may be used in any order.) You can use two `L`'s to get a `long long int` constant; add a `U` and you have an `unsigned long long int` constant.

There is no way to force an integer constant to be interpreted as a short integer. In addition, an integer constant will never be interpreted as a short integer by default, even if its value is small enough to be represented as one.

1.3.2 Character Constants

A character constant is usually a single character enclosed within single quotation marks, such as `'Q'`. A character constant is of type `int` by default.

Some characters, such as the single quotation mark character `'` itself, cannot be represented using only one character. To represent such characters, there are several "escape sequences" that you can use:

`\\` Backslash character.

`\?` Question mark character.

`\'` Single quotation mark.

`\"` Double quotation mark.

`\a` Audible alert.

`\b` Backspace character.

`\e` <ESC> character.

`\f` Form feed.

`\n` Newline character.

`\r` Carriage return.

`\t` Horizontal tab.

`\v` Vertical tab.

`\ooo` Octal number.

`\xhh` Hexadecimal number.

To use any of these escape sequences, enclose the sequence in single quotes, and treat it as if it were any other character. For example, the letter m is `'m'` and the newline character is `'\n'`.

The octal number escape sequence is the backslash character followed by one, two, or three octal digits (0 to 7). For example, 101 is the octal equivalent of 65, which is the ASCII character `'A'`. Thus, the character constant `'\101'` is the same as the character constant `'A'`.

The hexadecimal escape sequence is the backslash character, followed by `x` and an unlimited number of hexadecimal digits (0 to 9, and `a` to `f` or `A` to `F`).

While the number of possible hexadecimal digits is unlimited, the number of character constants is not. (The much-used extended ASCII character set has only 256 characters in it.) If you try to use a hexadecimal value that is outside the range of characters, you will get a compile-time error.

1.3.3 Real Number Constants

A real number constant is a value that represents a fractional (floating point) number. It consists of a sequence of digits which represents the integer (or "whole") part of the number, a decimal point, and a sequence of digits which represents the fractional part.

Either the integer part or the fractional part may be omitted, but not both. Here are some examples:

```
double a, b, c, d, e, f;

a = 4.7;      /* This is okay. */

b = 4.;       /* This is okay. */

c = 4;        /* This is okay. */

d = .7;       /* This is okay. */

e = 0.7;      /* This is okay. */

f = .;        /* This is NOT okay! */
```

Real number constants can also be followed by `e` or `E`, and an integer exponent. The exponent can be either positive or negative.

```
double x, y;

x = 5e2;   /* x is 5 x (10 ^ 2), or 500.0. */
y = 5e-2;  /* y is 5 x (10 ^ -2), or 0.05. */
```

You can append a letter to the end of a real number constant to cause it to be of a particular type. If you append the letter F (or f) to a real number constant, then its type is `float`. If you append the letter L (or l), then its type is `long double`. If you do not append any letters, then its type is `double`.

1.3.4 String Constants

A string constant is a sequence of characters, digits, and/or escape sequences enclosed within double quotation marks. A string constant is of type "array of characters". All string constants contain a null termination character (`\0`) as their last character. Strings are stored as arrays of characters, with no inherent size attribute. The null termination character lets string-processing functions know where the string ends.

Adjacent string constants are concatenated (combined) into one string, with the null termination character added to the end of the final concatenated string.

A string cannot contain double quotation marks, as double quotation marks are used to enclose the string. To include the double quotation mark character in a string, use the `\"` escape sequence. You can use any of the escape sequences that can be used as character constants in strings. Here are some example of string constants:

```
/* This is a single string constant. */
"tutti frutti ice cream"

/* These string constants will be concatenated, same as above. */
"tutti " "frutti" " ice " "cream"

/* This one uses two escape sequences. */
"\"hello, world!\""
```

If a string is too long to fit on one line, you can use a backslash `\` to break it up onto separate lines.

```
"Today's special is a pastrami sandwich on rye bread with \
a potato knish and a cherry soda."
```

Adjacent strings are automatically concatenated, so you can also have string constants span multiple lines by writing them as separate, adjacent, strings. For example:

```
"Tomorrow's special is a corned beef sandwich on "
"pumpernickel bread with a kasha knish and seltzer water."
```

is the same as

```
"Tomorrow's special is a corned beef sandwich on \
pumpernickel bread with a kasha knish and seltzer water."
```

To insert a newline character into the string, so that when the string is printed it will be printed on two different lines, you can use the newline escape sequence '`\n`'.

```
printf ("potato\nknish");
```

prints

```
potato
knish
```

1.4 Operators

An operator is a special token that performs an operation, such as addition or subtraction, on either one, two, or three operands. Full coverage of operators can be found in a later chapter. See Chapter 3 [Expressions and Operators], page 24.

1.5 Separators

A separator separates tokens. White space (see next section) is a separator, but it is not a token. The other separators are all single-character tokens themselves:

```
( ) [ ] { } ; , . :
```

1.6 White Space

White space is the collective term used for a number of characters, including the space character, the tab character, and the newline character. In C programs, white space is ignored (outside of string and character constants), and is therefore optional, except when it is used to separate tokens. This means that

```
#include <stdio.h>

int
main()
{
  printf( "hi there\n" );
  return 0;
}
```

and

```
#include <stdio.h> int main(){printf("hi there\n");
return 0;}
```

are functionally the same program.

Although you must use white space to separate many tokens, no white space is required between operators and operands, nor is it required between other separators and that which they separate.

```
/* All of these are valid. */

x++;
x ++ ;
x=y+z;
x = y + z ;
x=array[2];
x = array [ 2 ] ;
fraction=numerator/*denominator_ptr;
fraction = numerator / * denominator_ptr ;
```

Furthermore, wherever one space is allowed, any amount of white space is allowed.

```
/* These two statements are functionally identical. */
x++;

x
       ++       ;
```

In string constants, spaces and tabs are not ignored; rather, they are part of the string. Therefore,

```
"potato knish"
```

is not the same as

```
"potato                           knish"
```

2 Data Types

2.1 Primitive Data Types

2.1.1 Integer Types

The integer data types range in size from at least 8 bits to at least 64 bits. You should use them for storing whole number values (and the `char` data type for storing characters). (Note that the sizes and ranges listed for these types are minimums; depending on your computer platform, these sizes and ranges may be larger.)

- `signed char`
  The 8-bit `signed char` data type can hold integer values in the range of −128 to 127.
- `unsigned char`
  The 8-bit `unsigned char` data type can hold integer values in the range of 0 to 255.
- `char`
  Depending on your system, the `char` data type is defined as equivalent to either the `signed char` or the `unsigned char` data type. By convention, you should use the `char` data type specifically for storing ASCII characters (such as 'm'), including escape sequences (such as '\n').
- `short int`
  The 16-bit `short int` data type can hold integer values in the range of −32,768 to 32,767. You may also refer to this data type as `short`, `signed short int`, or `signed short`.
- `unsigned short int`
  The 16-bit `unsigned short int` data type can hold integer values in the range of 0 to 65,535. You may also refer to this data type as `unsigned short`.
- `int`
  The 32-bit `int` data type can hold integer values in the range of −2,147,483,648 to 2,147,483,647. You may also refer to this data type as `signed int` or `signed`.
- `unsigned int`
  The 32-bit `unsigned int` data type can hold integer values in the range of 0 to 4,294,967,295. You may also refer to this data type simply as `unsigned`.
- `long int`
  The 32-bit `long int` data type can hold integer values in the range of at least −2,147,483,648 to 2,147,483,647. (Depending on your system, this data type might be 64-bit, in which case its range is identical to that of the `long long int` data type.) You may also refer to this data type as `long`, `signed long int`, or `signed long`.
- `unsigned long int`
  The 32-bit `unsigned long int` data type can hold integer values in the range of at least 0 to 4,294,967,295. (Depending on your system, this data type might be 64-bit, in which case its range is identical to that of the `unsigned long long int` data type.) You may also refer to this data type as `unsigned long`.
- `long long int`
  The 64-bit `long int` data type can hold integer values in the range of -

9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. You may also refer to this data type as `long long`, `signed long long int` or `signed long long`.

- `unsigned long long int`
  The 64-bit `unsigned long long int` data type can hold integer values in the range of at least 0 to 18,446,744,073,709,551,615. You may also refer to this data type as `unsigned long long`.

2.1.2 Real Number Types

There are several data types that represent fractional numbers, such as 3.14159 and 2.83. The exact sizes and ranges for the floating point data types can vary from system to system; these values are stored in macro definitions in the library header file `float.h`. In this section, we include the names of the macro definitions in place of their possible values:

- `float`
  The `float` data type is the smallest of the three floating point types, if they differ in size at all. Its minimum value is stored in the `FLT_MIN`, and is should be no greater than 1e($-$37). Its maximum value is stored in `FLT_MAX`, and is should be no less than 1e37.
- `double`
  The `double` data type is at least as large as the `float` type, and it may be larger. Its minimum value is stored in `DBL_MIN`, and its maximum value is stored in `DBL_MAX`.
- `long double`
  The `long double` data type is at least as large as the `float` type, and it may be larger. Its minimum value is stored in `LDBL_MIN`, and its maximum value is stored in `LDBL_MAX`.

All floating point data types are signed; trying to use `unsigned float`, for example, will cause a compile-time error.

2.2 Enumerations

An enumeration is a custom data type used for storing constant integer values and referring to them by names. By default, these values are of type `signed int`; however, you can use the `-fshort-enums` GCC compiler option to cause the smallest possible integer type to be used instead.

2.2.1 Defining Enumerations

You define an enumeration using the `enum` keyword, followed by the name of the enumeration (this is optional), followed by a list of constant names (separated by commas and enclosed in braces), and ending with a semicolon.

```
enum fruit {grape, cherry, lemon, kiwi};
```

That example defines an enumeration, `fruit`, which contains four constant integer values, `grape`, `cherry`, `lemon`, and `kiwi`, whose values are, by default, 0, 1, 2, and 3, respectively. You can also specify one or more of the values explicitly:

```
enum more_fruit {banana = -17, apple, blueberry, mango};
```

That example defines `banana` to be -17, and the remaining values are incremented by 1—`apple` is -16, `blueberry` is -15, and `mango` is -14. Unless specified otherwise, an enumeration value is equal to one more than the previous value (and the first value defaults to 0).

You can also refer to an enumeration value defined earlier in the same enumeration:

```
enum yet_more_fruit {kumquat, raspberry, peach,
                     plum = peach + 2};
```

In that example, `kumquat` is 0, `raspberry` is 1, `peach` is 2, and `plum` is 4.

2.2.2 Declaring Enumerations

You can declare variables of an enumeration type both when the enumeration is defined and afterward. This example declares one variable, named `my_fruit` of type `enum fruit`:

```
enum fruit {banana, apple, blueberry, mango} my_fruit;
```

While this example declares the type and variable separately:

```
enum fruit {banana, apple, blueberry, mango};
enum fruit my_fruit;
```

(Of course, you couldn't declare it that way if you hadn't named the enumeration.)

Although such variables are considered to be of an enumeration type, you can assign them any value that you could assign to an `int` variable, including values from other enumerations. Furthermore, any variable that can be assigned an `int` value can be assigned a value from an enumeration.

However, you cannot change the values in an enumeration once it has been defined; they are constant values. For example, this won't work:

```
enum fruit {banana, apple, blueberry, mango};
banana = 15;  /* You can't do this! */
```

2.3 Unions

A union is a custom data type used for storing several variables in the same memory space. Although you can access any of those variables at any time, you should only read from one of them at a time—assigning a value to one of them overwrites the values in the others.

2.3.1 Defining Unions

You define a union using the `union` keyword followed by the declarations of the union's members, enclosed in braces. You declare each member of a union just as you would normally declare a variable—using the data type followed by one or more variable names separated by commas, and ending with a semicolon. Then end the union definition with a semicolon after the closing brace.

You should also include a name for the union between the `union` keyword and the opening brace. This is syntactically optional, but if you leave it out, you can't refer to that union data type later on (without a `typedef`, see Section 4.14 [The typedef Statement], page 44).

Here is an example of defining a simple union for holding an integer value and a floating point value:

```
union numbers
  {
    int i;
    float f;
  };
```

That defines a union named `numbers`, which contains two members, `i` and `f`, which are of type `int` and `float`, respectively.

2.3.2 Declaring Union Variables

You can declare variables of a union type when both you initially define the union and after the definition, provided you gave the union type a name.

2.3.2.1 Declaring Union Variables at Definition

You can declare variables of a union type when you define the union type by putting the variable names after the closing brace of the union definition, but before the final semicolon. You can declare more than one such variable by separating the names with commas.

```
union numbers
  {
    int i;
    float f;
  } first_number, second_number;
```

That example declares two variables of type `union numbers`, `first_number` and `second_number`.

2.3.2.2 Declaring Union Variables After Definition

You can declare variables of a union type after you define the union by using the `union` keyword and the name you gave the union type, followed by one or more variable names separated by commas.

```
union numbers
  {
    int i;
    float f;
  };
union numbers first_number, second_number;
```

That example declares two variables of type `union numbers`, `first_number` and `second_number`.

2.3.2.3 Initializing Union Members

You can initialize the first member of a union variable when you declare it:

```
union numbers
  {
    int i;
    float f;
  };
union numbers first_number = { 5 };
```

In that example, the `i` member of `first_number` gets the value 5. The `f` member is left alone.

Another way to initialize a union member is to specify the name of the member to initialize. This way, you can initialize whichever member you want to, not just the first one. There are two methods that you can use—either follow the member name with a colon, and then its value, like this:

```
union numbers first_number = { f: 3.14159 };
```

or precede the member name with a period and assign a value with the assignment operator, like this:

```
union numbers first_number = { .f = 3.14159 };
```

You can also initialize a union member when you declare the union variable during the definition:

```
union numbers
  {
    int i;
    float f;
  } first_number = { 5 };
```

2.3.3 Accessing Union Members

You can access the members of a union variable using the member access operator. You put the name of the union variable on the left side of the operator, and the name of the member on the right side.

```
union numbers
  {
    int i;
    float f;
  };
union numbers first_number;
first_number.i = 5;
first_number.f = 3.9;
```

Notice in that example that giving a value to the `f` member overrides the value stored in the `i` member.

2.3.4 Size of Unions

This size of a union is equal to the size of its largest member. Consider the first union example from this section:

```
union numbers
  {
    int i;
    float f;
  };
```

The size of the union data type is the same as `sizeof (float)`, because the `float` type is larger than the `int` type. Since all of the members of a union occupy the same memory space, the union data type size doesn't need to be large enough to hold the sum of all their sizes; it just needs to be large enough to hold the largest member.

2.4 Structures

A structure is a programmer-defined data type made up of variables of other data types (possibly including other structure types).

2.4.1 Defining Structures

You define a structure using the `struct` keyword followed by the declarations of the structure's members, enclosed in braces. You declare each member of a structure just as you would normally declare a variable—using the data type followed by one or more variable names separated by commas, and ending with a semicolon. Then end the structure definition with a semicolon after the closing brace.

You should also include a name for the structure in between the `struct` keyword and the opening brace. This is optional, but if you leave it out, you can't refer to that structure data type later on (without a `typedef`, see Section 4.14 [The typedef Statement], page 44).

Here is an example of defining a simple structure for holding the X and Y coordinates of a point:

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

That defines a structure type named `struct point`, which contains two members, `x` and `y`, both of which are of type `int`.

2.4.2 Declaring Structure Variables

You can declare variables of a structure type when both you initially define the structure and after the definition, provided you gave the structure type a name.

2.4.2.1 Declaring Structure Variables at Definition

You can declare variables of a structure type when you define the structure type by putting the variable names after the closing brace of the structure definition, but before the final semicolon. You can declare more than one such variable by separating the names with commas.

```
struct point
  {
    int x, y;
  } first_point, second_point;
```

That example declares two variables of type `struct point`, `first_point` and `second_point`.

2.4.2.2 Declaring Structure Variables After Definition

You can declare variables of a structure type after defining the structure by using the `struct` keyword and the name you gave the structure type, followed by one or more variable names separated by commas.

```
struct point
  {
    int x, y;
  };
struct point first_point, second_point;
```

That example declares two variables of type `struct point`, `first_point` and `second_point`.

2.4.2.3 Initializing Structure Members

You can initialize the members of a structure type to have certain values when you declare structure variables. One way is to specify the values in a set of braces and separated by commas. Those values are assigned to the structure members in the same order that the members are declared in the structure in definition.

```
struct point
  {
    int x, y;
  };
struct point first_point = { 5, 10 };
```

In that example, the `x` member of `first_point` gets the value 5, and the `y` member gets the value 10.

Another way to initialize the members is to specify the name of the member to initialize. This way, you can initialize the members in any order you like, and even leave some of them uninitialized. There are two methods that you can use—either follow the member name with a colon, and then its value, like this:

```
struct point first_point = { y: 10, x: 5 };
```

or precede the member name with a period and assign a value with the assignment operator, like this:

```
struct point first_point = { .y = 10, .x = 5 };
```

You can also initialize the structure variable's members when you declare the variable during the structure definition:

```
struct point
  {
    int x, y;
  } first_point = { 5, 10 };
```

You can also initialize fewer than all of a structure variable's members:

```
struct point
  {
    int x, y;
  };
struct point first_point = { 5 };
```

In that example, only one member, `x`, is initialized. `y` is left uninitialized. (This is because the members are initialized in order of declaration, and `x` was declared before `y`.)

Here is another example that initializes a structure's members which are structure variables themselves:

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

struct rectangle
  {
    struct point top_left, bottom_right;
  };

struct rectangle my_rectangle = { {0, 5}, {10, 0} };
```

That example defines the `rectangle` structure to consist of two `point` structure variables. Then it declares one variable of type `struct rectangle` and initializes its members. Since its members are structure variables, we used an extra set of braces surrounding the members that belong to the `point` structure variables. However, those extra braces are not necessary; they just make the code easier to read.

2.4.3 Accessing Structure Members

You can access the members of a structure variable using the member access operator. You put the name of the structure variable on the left side of the operator, and the name of the member on the right side.

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

struct point first_point;

first_point.x = 0;
first_point.y = 5;
```

You can also access the members of a structure variable which is itself a member of a structure variable.

```
struct rectangle
  {
    struct point top_left, bottom_right;
  };

struct rectangle my_rectangle;

my_rectangle.top_left.x = 0;
my_rectangle.top_left.y = 5;

my_rectangle.bottom_right.x = 10;
my_rectangle.bottom_right.y = 0;
```

2.4.4 Bit Fields

You can create structures with integer members of nonstandard sizes, called *bit fields*. You do this by specifying an integer (`int`, `char`, `long int`, etc.) member as usual, and inserting a colon and the number of bits that the member should occupy in between the member's name and the semicolon.

```
struct card
  {
    unsigned int suit : 2;
    unsigned int face_value : 4;
  };
```

That example defines a structure type with two bit fields, `suit` and `face_value`, which take up 2 bits and 4 bits, respectively. `suit` can hold values from 0 to 3, and `face_value` can hold values from 0 to 15. Notice that these bit fields were declared as `unsigned int`; had they been signed integers, then their ranges would have been from -2 to 1, and from -8 to 7, respectively.

More generally, the range of an unsigned bit field of N bits is from 0 to $2^N - 1$, and the range of a signed bit field of N bits is from $-(2^N)/2$ to $((2^N)/2) - 1$.

2.4.5 Size of Structures

The size of a structure type is equal to the sum of the size of all of its members, possibly including padding to cause the structure type to align to a particular byte boundary. The details vary depending on your computer platform, but it would not be atypical to see structures padded to align on four- or eight-byte boundaries. This is done in order to speed up memory accesses of instances of the structure type.

If you wish to explicitly omit padding from your structure types (which may, in turn, decrease the speed of structure memory accesses), then GCC provides multiple methods of turning packing off. The quick and easy method is to use the `-fpack-struct` compiler option. For more details on omitting packing, please see the GCC manual which corresponds to your version of the compiler.

2.5 Arrays

An array is a data structure that lets you store one or more elements consecutively in memory. In C, array elements are indexed beginning at position zero, not one.

2.5.1 Declaring Arrays

You declare an array by specifying the data type for its elements, its name, and the number of elements it can store. Here is an example that declares an array that can store ten integers:

```
int my_array[10];
```

The number of elements in an array must be positive.

2.5.2 Initializing Arrays

You can initialize the elements in an array when you declare it by listing the initializing values, separated by commas, in a set of braces. Here is an example:

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

You don't have to initialize all of the array elements. For example, this code initializes only the first three elements:

```
int my_array[5] = { 0, 1, 2 };
```

That leaves the last two elements uninitialized.

2.5.3 Accessing Array Elements

You can access the elements of an array by specifying the array name, followed by the element index, enclosed in brackets. Remember that the array elements are numbered starting with zero. Here is an example:

```
my_array[0] = 5;
```

That assigns the value 5 to the first element in the array, at position zero. You can treat individual array elements like variables of whatever data type the array is made up of. For example, if you have an array made of a structure data type, you can access the structure elements like this:

```
struct point
{
  int x, y;
};
struct point point_array[2] = { {4, 5}, {8, 9} };
point_array[0].x = 3;
```

2.5.4 Multidimensional Arrays

You can make multidimensional arrays, or "arrays of arrays". You do this by adding an extra set of brackets and array lengths for every additional dimension you want your array to have. For example, here is a declaration for a two-dimensional array that holds five elements in each dimension (a two-element array consisting of five-element arrays):

```
int two_dimensions[2][5] { {1, 2, 3, 4, 5}, {6, 7, 8, 9, 10} };
```

Multidimensional array elements are accessed by specifying the desired index of both dimensions:

```
two_dimensions[1][3] = 12;
```

2.5.5 Arrays as Strings

You can use an array of characters to hold a string (see Section 1.3.4 [String Constants], page 5). The array may be built of either signed or unsigned characters.

When you declare the array, you can specify the number of elements it will have. That number will be the maximum number of characters that should be in the string, including the null character used to end the string. If you choose this option, then you do not have to initialize the array when you declare it. Alternately, you can simply initialize the array to a value, and its size will then be exactly large enough to hold whatever string you used to initialize it.

There are two different ways to initialize the array. You can specify of comma-delimited list of characters enclosed in braces, or you can specify a string literal enclosed in double quotation marks.

Here are some examples:

```
char blue[26];
char yellow[26] = {'y', 'e', 'l', 'l', 'o', 'w', '\0'};
char orange[26] = "orange";
char gray[] = {'g', 'r', 'a', 'y', '\0'};
char salmon[] = "salmon";
```

In each of these cases, the null character `\0` is included at the end of the string, even when not explicitly stated. (Note that if you initialize a string using an array of individual characters, then the null character is *not* guaranteed to be present. It might be, but such an occurrence would be one of chance, and should not be relied upon.)

After initialization, you cannot assign a new string literal to an array using the assignment operator. For example, this *will not work*:

```
char lemon[26] = "custard";
lemon = "steak sauce";      /* Fails! */
```

However, there are functions in the GNU C library that perform operations (including copy) on string arrays. You can also change one character at a time, by accessing individual string elements as you would any other array:

```
char name[] = "bob";
name[0] = 'r';
```

It is possible for you to explicitly state the number of elements in the array, and then initialize it using a string that has more characters than there are elements in the array. This is not a good thing. The larger string will *not* override the previously specified size of the array, and you will get a compile-time warning. Since the original array size remains, any part of the string that exceeds that original size is being written to a memory location that was not allocated for it.

2.5.6 Arrays of Unions

You can create an array of a union type just as you can an array of a primitive data type.

```
union numbers
  {
    int i;
    float f;
  };
union numbers number_array [3];
```

That example creates a 3-element array of `union numbers` variables called `number_array`. You can also initialize the first members of the elements of a number array:

```
struct point point_array [3] = { {3}, {4}, {5} };
```

The additional inner grouping braces are optional.

After initialization, you can still access the union members in the array using the member access operator. You put the array name and element number (enclosed in brackets) to the left of the operator, and the member name to the right.

```
union numbers number_array [3];
number_array[0].i = 2;
```

2.5.7 Arrays of Structures

You can create an array of a structure type just as you can an array of a primitive data type.

```
struct point
  {
    int x, y;
  };
struct point point_array [3];
```

That example creates a 3-element array of `struct point` variables called `point_array`. You can also initialize the elements of a structure array:

```
struct point point_array [3] = { {2, 3}, {4, 5}, {6, 7} };
```

As with initializing structures which contain structure members, the additional inner grouping braces are optional. But, if you use the additional braces, then you can partially initialize some of the structures in the array, and fully initialize others:

```
struct point point_array [3] = { {2}, {4, 5}, {6, 7} };
```

In that example, the first element of the array has only its `x` member initialized. Because of the grouping braces, the value 4 is assigned to the `x` member of the second array element, *not* to the `y` member of the first element, as would be the case without the grouping braces.

After initialization, you can still access the structure members in the array using the member access operator. You put the array name and element number (enclosed in brackets) to the left of the operator, and the member name to the right.

```
struct point point_array [3];
point_array[0].x = 2;
point_array[0].y = 3;
```

2.6 Pointers

Pointers hold memory addresses of stored constants or variables. For any data type, including both primitive types and custom types, you can create a pointer that holds the memory address of an instance of that type.

2.6.1 Declaring Pointers

You declare a pointer by specifying a name for it and a data type. The data type indicates of what type of variable the pointer will hold memory addresses.

To declare a pointer, include the indirection operator (see Section 3.2.8 [The Indirection Operator], page 26) before the identifier. Here is the general form of a pointer declaration:

```
data-type * name;
```

You can also put the operator either directly next to *name*, or directly next to *data-type*, like these:

```
data-type *name;
data-type* name;
```

Any of these three is fine, and they all work the same way (white space is not significant, as usual). Here is an example of declaring a pointer to hold the address of an `int` variable:

```
int *ip;
```

Be careful, though: when declaring multiple pointers in the same statement, you must explicitly declare each as a pointer, using the indirection operator:

```
int *bob, *emily;  /* Two pointers. */
int *rob, laura;   /* A pointer and an integer variable. */
```

2.6.2 Initializing Pointers

You can initialize a pointer when you first declare it by specifying a variable address to store in it. For example, the following code declares an `int` variable '`i`', and a pointer which is initialized with the address of '`i`':

```
int i;
int *ip = &i;
```

Note the use of the address operator (see Section 3.2.7 [The Address Operator], page 26), used to get the memory address of a variable. *Be careful*, though: after you declare a pointer, you do *not* use the indirection operator with the pointer's name when assigning it a new address to point to. On the contrary, that would change the value of the variable that the points to, not the value of the pointer itself. For example:

```
int i, j;
int *ip = &i;  /* 'ip' now holds the address of 'i'. */
ip = &j;       /* 'ip' now holds the address of 'j'. */
*ip = &i;      /* 'j' now holds the address of 'i'. */
```

The value stored in a pointer is an integral number: a location within the computer's memory space. If you are so inclined, you can assign pointer values explicitly using literal integers, casting them to the appropriate pointer type. However, we do not recommend this practice unless you need to have extremely fine-tuned control over what is stored in memory, and you know exactly what you are doing. It would be all too easy to accidentally overwrite something that you did not intend to.

2.6.3 Pointers to Unions

You can create a pointer to a union type just as you can a pointer to a primitive data type.

```
union numbers
  {
    int i;
    float f;
  };
union numbers foo = {4};
union numbers *number_ptr = &foo;
```

That example creates a new union type, `union numbers`, and declares (and initializes the first member of) a variable of that type named `foo`. Finally, it declares a pointer to the type `union numbers`, and gives it the address of `foo`.

You can access the members of a union variable through a pointer, but you can't use the regular member access operator anymore. Instead, you have to use the indirect member access operator (see Section 3.3.14 [Member Access Operators], page 34). Continuing with the previous example, the following example will change the value of the first member of `foo`:

```
number_ptr -> i = 450;
```

Now the i member in foo is 450.

2.6.4 Pointers to Structures

You can create a pointer to a structure type just as you can a pointer to a primitive data type.

```
struct fish
  {
    float length, weight;
  };
struct fish salmon = {4.3, 5.8};
struct fish *fish_ptr = &salmon;
```

That example creates a new structure type, struct fish, and declares (and initializes) a variable of that type named salmon. Finally, it declares a pointer to the type struct fish, and gives it the address of salmon.

You can access the members of a structure variable through a pointer, but you can't use the regular member access operator anymore. Instead, you have to use the indirect member access operator (see Section 3.3.14 [Member Access Operators], page 34). Continuing with the previous example, the following example will change the values of the members of salmon:

```
fish_ptr -> length = 5.1;
fish_ptr -> weight = 6.2;
```

Now the length and width members in salmon are 5.1 and 6.2, respectively.

2.7 Incomplete Types

You can define structures, unions, and enumerations without listing their members (or values, in the case of enumerations). Doing so results in an incomplete type. You can't declare variables of incomplete types, but you can work with pointers to those types.

```
struct point;
```

At some time later in your program you will want to complete the type. You do this by defining it as you usually would:

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

2.8 Type Specifiers

There are two type specifiers that you can prepend to your variable declarations which change how the variables may be accessed: const and volatile.

const causes the variable to be read-only; after initialization, its value may not be changed.

```
const float pi = 3.14159;
```

In addition to helping to prevent accidental value changes, declaring variables with `const` can aid the compiler in code optimization.

`volatile` tells the compiler that the variable is explicitly changeable, and seemingly useless accesses of the variable (for instance, via pointers) should not be optimized away. You might use `volatile` variables to store data that is updated via callback functions.

```
volatile float currentTemperature = 40.0;
```

2.9 Storage Class Specifiers

There are four storage class specifiers that you can prepend to your variable declarations which change how the variables are stored in memory: `auto`, `extern`, `register`, and `static`.

You use `auto` for variables which are local to a function, and whose values should be discarded upon return from the function in which they are declared. This is the default behavior for variables declared within functions.

```
void
foo (int value)
{
  auto int x = value;
  ...
  return;
}
```

`register` is nearly identical in purpose to `auto`, except that it also suggests to the compiler that the variable will be heavily used, and, if possible, should be stored in a register in memory. You cannot use the address-of operator to obtain the address of a variable declared with `register`.

`static` is essentially the opposite of `auto`: when applied to variables within a function or block, these variables will retain their value even when the function or block is finished.

```
int
sum (int x)
{
  static int sumSoFar = 0;
  sumSoFar = sumSoFar + x;
  return x;
}
```

You can also declare variables outside of functions to be `static`; such variables are visible (global) to the current source file (but not other source files).

`extern` is useful for declaring variables that you want to be visible to all source files that are linked into your project. You cannot initialize a variable in an `extern` declaration, as no space is actually allocated during the declaration. You must make both an `extern` declaration (typically in a header file that is included by the other source files which need to access the variable) and a non-`extern` declaration which is where space is actually allocated to store the variable. The `extern` declaration may be repeated multiple times.

```
extern int numberOfClients;

...

int numberOfClients = 0;
```

See Chapter 6 [Program Structure and Scope], page 52, for related information.

2.10 Renaming Types

Sometimes it is conveninent to give a new name to a type. You can do this using the `typedef` statement. See Section 4.14 [The typedef Statement], page 44, for more information.

3 Expressions and Operators

3.1 Expressions

An *expression* consists of at least one operand and zero or more operators. The operands may be any value, including constants, variables, and function calls that return values. Here are some examples:

```
47
2 + 2
function()
```

The last of those, `function()`, is only an expression if `function()` has a return type other than `void`.

You can use parentheses to group subexpressions:

```
( 2 * ( ( 3 + 10 ) - ( 2 * 6 ) ) )
```

Innermost expressions are evaluated first. In the above example, `3 + 10` and `2 * 6` evaluate to `13` and `12`, respectively. Then `12` is subtracted from `13`, resulting in `1`. Finally, `1` is multiplied by `2`, resulting in `2`. The outermost parentheses are completely optional.

An *operator* specifies an operation to be performed on its operand(s). Operators may have one, two, or three operands, depending on the operator.

3.2 Unary Operators

Unary operators perform an operation on a single operand.

3.2.1 The Increment Operator

The increment operator `++` adds 1 to its operand. The operand must be a either a variable of one of the primitive data types, a pointer, or an enumeration variable. Here are some examples:

```
int x = 5;
char y = 'B';
float z = 5.2;
int *p = &x;

x++;   /* x is now 6. */
y++;   /* y is now 'C'. */
z++;   /* z is now 6.2. */
p++;   /* p is now &x + sizeof(int). */
```

You can use the increment operator either before or after the operand. A prefix increment adds 1 before the operand is evaluated. A postfix increment adds 1 after the operand is evaluated. In the previous examples, that wouldn't have made any difference. However, there are cases where it does make a difference:

```
int x = 5;
printf ("%d \n", x++);    /* Print x and then increment it. */
/* x is now equal to 6. */
printf ("%d \n", ++x);    /* Increment x and then print it. */
```

The output of the above example is:

```
5
7
```

3.2.2 The Decrement Operator

The decrement operator `--` subtracts 1 from its operand. The operand must be a either a variable of one of the primitive data types, a pointer, or an enumeration variable. Here are some examples:

```
int x = 5;
char y = 'B';
float z = 5.2;
int *p = &x;

x--;    /* x is now 4. */
y--;    /* y is now ‘A’. */
z--;    /* z is now 4.2. */
p--;    /* p is now &x - sizeof(int). */
```

You can use the decrement operator either before or after the operand. A prefix decrement subtracts 1 before the operand is evaluated. A postfix increment subtracts 1 after the operand is evaluated. In the previous examples, that wouldn't have made any difference. However, there are cases where it does make a difference:

```
int x = 5;
printf ("%d \n", x--);    /* Print x and then decrement it. */
/* x is now 4 */
printf ("%d \n", --x);    /* Decrement x and then print it. */
```

The output of the above example is:

```
5
3
```

3.2.3 The Positive Operator

You can use the positive operator `+` on numeric values to indicate that their value is positive. By default, values are positive unless explicitly stated to be negative, so there is no need to use this operator as far as the compiler is concerned. However, you can use it to visually reinforce the fact that a value is positive. Here are some examples:

```
int x = +5;
float y = +3.14159;
```

3.2.4 The Negative Operator

You can use the negative operator `-` on numeric variables and constants to indicate that their value is negative. Here are some examples:

```
int x = -5;
float y = -3.14159;
```

If the operand you use with the negative operator is of an unsigned data type, then the result is not negative, but rather the maximum value of the unsigned data type, minus the value of the operand.

3.2.5 The Logical Negation Operator

You can use the logical negation operator `!` to get the logical opposite of its operand. If its operand is 0 (or null, if the operand is a pointer), then the result of the logical negation operator is 1. If its operand is anything other than 0 (or null), then the result of the logical negation operator is 0. In any case, the result is an integer value. Here are some examples:

```
int x = !5;   /* x is 0. */
if (!x)
  printf ("x is 0");
```

3.2.6 The Bitwise Complement Operator

You can use the bitwise complement operator `~` to get the one's complement of its operand. The operand must be an integer or character type. The bitwise complement operator examines its operand's bits, and changes all 0 bits to 1 and all 1 bits to 0. Here is an example:

```
unsigned int x = 500;
unsigned int y;
y = ~x;
```

Using signed data types with the bitwise complement operator may cause portability problems, so use unsigned data types for maximum portability.

3.2.7 The Address Operator

You can use the address operator `&` to obtain the memory address of its operand. You can use this operator both with variables of any data type (including arrays and structures) and with functions, but you can't use it with literal values. You should only store the result of the address operator in pointer variables.

```
int x = 5;
int *ptr = &x;
```

3.2.8 The Indirection Operator

You can use the indirection operator `*` to obtain the value stored at the address specified by its operand. This is known as *dereferencing* its operand. Its operand must be a pointer.

```
int x = 5;
int y;
int *ptr;

ptr = &x;     /* ptr now holds the address of x. */

y = *ptr;     /* y gets the value stored at the address
                 stored in ptr. */
```

The result of using the indirection operator with pointers that have not been initialized is unspecified; usually the program will crash.

3.2.9 The sizeof Operator

You can use the `sizeof` operator to obtain the size (in bytes) of the data type of its operand. The operand may be an actual type specifier (such as `int` or `float`), as well as any valid expression. You must enclose the operand in parentheses after the operator. Here are some examples:

```
size_t a = sizeof(int);
size_t b = sizeof(float);
size_t c = sizeof(5);
size_t d = sizeof(5.143);
```

The result of the `sizeof` operator is of a type called `size_t`, which is defined in the header file `<stddef.h>`. `size_t` is an unsigned integer type, perhaps identical to `unsigned int` or `unsigned long int`; it varies from system to system.

3.2.10 Type Casts

You can use a type cast to explicitly cause an expression to be of a specified data type. A type cast consists of a type specifier enclosed in parentheses, followed by an expression. To ensure proper casting, you should also enclose the expression that follows the type specifier in parentheses. Here is an example:

```
float x;
int y = 7;
int z = 3;
x = (float) (y / z);
```

In that example, since `y` and `z` are both integers, integer division is performed, and even though `x` is a floating-point variable, it receives the value 2. By explicitly casting the result of the division to `float`, the floating-point value 2.333... is retained and assigned to `x`.

Type casting also works with custom data types. Here is an example of converting an array of 8 bytes to a 8-byte structure type:

```
struct fooType
{
  float f;
  unsigned short int a;
  unsigned short int b;
};

struct fooType foo;
unsigned char byteArray[8];

foo = (struct fooType) byteArray;
```

In practice, you should also pack your structure type to ensure that the compiler doesn't add any padding, thereby increasing the size of the type. In general, different compilers may use different representations for data types, so such constructs are not portable.

3.2.11 Array Subscripts

You can access array elements by specifying the name of the array, and the array subscript (or index, or element number) enclosed in brackets. Here is an example, supposing an integer array called `my_array`:

```
my_array[0] = 5;
```

3.2.12 Function Calls as Expressions

A call to any function which returns a value is an expression.

```
int function(void);
...
a = 10 + function();
```

3.3 Binary Operators

3.3.1 The Addition Operator

You use the addition operator `+` to add two operands. You put the operands on either side of the operator, and it does not matter which operand goes on which side (in the absence of side effects): `3 + 5` and `5 + 3` both result in 8. The operands must be either expressions of a primitive data type or pointers.

```
x = 5 + 3;
y = 10 + 37;
z = 1 + 2 + 3 + 4 + 5;
```

When you use more than one addition operator (and more than two operands), such as in the last example, the expression is evaluated from left to right.

3.3.2 The Subtraction Operator

You use the subtraction operator `-` to subtract its second operand from its first operand. You put the operands on either side of the operator, and it does matter which operand goes on which side: `3 - 5` and `5 - 3` do not have the same result. The operands must be either expressions of a primitive data type or pointers.

```
x = 5 - 3;
y = 57 - 10;
z = 5 - 4 - 3 - 2 - 1;
```

When you use more than one subtraction operator (and more than two operands), such as in the last example, the expression is evaluated from left to right.

3.3.3 The Multiplication Operator

You use the multiplication operator `*` to multiply two operands together. You put the operands on either side of the operator, and it does not matter which operand goes on which side: `3 * 5` and `5 * 3` both result in 15. The operands must be expressions of a primitive data type.

```
x = 5 * 3;
y = 47 * 1;
z = 1 * 2 * 3 * 4 * 5;
```

When you use more than one multiplication operator (and more than two operands), such as in the last example, the expression is evaluated from left to right.

3.3.4 The Division Operator

You use the division operator `/` to divide its first operand by its second operand. You put the operands on either side of the operator, and it does matter which operand goes on which side: `3 / 5` and `5 / 3` do not have the same result. The operands must be expressions of a primitive data type.

```
x = 5 / 3;
y = 940 / 20;
z = 100 / 2 / 2;
```

When you use more than one division operator (and more than two operands), such as in the last example, the expression is evaluated from left to right.

3.3.5 The Modulus Operator

You use the modulus operator `%` to obtain the remainder produced by dividing its two operands. You put the operands on either side of the operator, and it does matter which operand goes on which side: `3 % 5` and `5 % 3` do not have the same result. The operands must be expressions of a primitive data type.

```
x = 5 % 3;
y = 74 % 47;
z = 47 % 32 % 21;
```

When you use more than one modulus operator (and more than two operands), like in the last example, the expression is evaluated from left to right.

A common application of the modulus operator is to determine if one number is divisible by another number. If it is divisible, then the remainder is zero. Here is an example of that:

```
int counter;
for (counter = 0; counter <= 100; counter++)
  {
    if (counter % 5 == 0)
      printf ("%d\n", counter);
  }
```

That prints all of the integers from 0 to 100 that are divisible by 5.

3.3.6 The Shift Operators

You use the left-shift operator `<<` to shift its first operand's bits to the left. You specify the number of bit-places shifted with the second operand. If there is a 1 bit in the leftmost bit position, it will be discarded. New bits that are added to the rightmost bit position will all be 0.

```
x = 47;     /* 47 is 00101111 in binary. */
x << 1;     /* 00101111 << 1 is 01011110. */
```

You use the right-shift operator `>>` to shift its first operand's bits to the right. You specify the number of bit-places shifted with the second operand. If there is a 1 bit in the rightmost bit position, it will be discarded. New bits that are added to the leftmost bit position may be either 0 or 1. If the first operand is unsigned, then the added bits will be 0. If it is signed, the added bits will be either 0 *or* whatever value was previously in the leftmost bit position.

```
x = 47;    /* 47 is 00101111 in binary. */
x >> 1;    /* 00101111 >> 1 is 00010111. */
```

3.3.7 The Bitwise AND Operator

The bitwise AND operator & examines each bit in its two operands, and when two corresponding bits are both 1, the resulting bit is 1. In every other case the result is 0. Here is an example of how this operator works, using binary numbers:

```
11001001 & 10011011 = 10001001
```

If you look closely at that, you'll see that when a bit is 1 in both operands, the corresponding bit in the result is set to 1. Otherwise it is set to 0. Here is another example, this time in C:

```
char x = 149, y = 34, z;
z = x & y;
```

3.3.8 The Bitwise Inclusive OR Operator

The bitwise inclusive OR operator | examines each bit in its two operands, and when two corresponding bits are both 0, the resulting bit is 0. In every other case the resulting bit is 1. Here is an example of how this operator works, using binary numbers:

```
11001001 | 10011011 = 11011011
```

Here is another example, this time in C:

```
char x = 149, y = 34, z;
z = x | y;
```

3.3.9 The Bitwise Exclusive OR Operator

The bitwise exclusive OR operator ^ (also known as XOR) examines each bit in its two operands, and when two corresponding bits are different, the resulting bit is 1. When they are the same, the resulting bit is 0. Here is an example of how this operator works, using binary numbers:

```
11001001 | 10011011 = 01011001
```

Here is another example, this time in C.

```
char x = 149, y = 34, z;
z = x ^ y;
```

3.3.10 The Comparison Operators

You use the comparison operators to determine how two operands relate to each other: are they equal to each other, is one larger than the other, is one smaller than the other, and so on. When you use any of the comparison operators, the result is either 1 or 0, meaning true or false, respectively.

In the following code examples, the variables `x` and `y` stand for any two expressions of primitive types, or pointers.

3.3.10.1 The Equal-to Operator

Use the equal-to operator `==` to test two operands for equality. It evaluates to 1 if the two operands are equal, and 0 if the two operands are not equal.

```
if (x == y)
  puts ("x is equal to y");
else
  puts ("x is not equal to y");
```

3.3.10.2 The Not-Equal-to Operator

Use the not-equal-to operator `!=` to test two operands for inequality. If the two operands are not equal, the result is 1. Otherwise, if the two operands *are* equal, the result is 0.

```
if (x != y)
  puts ("x is not equal to y");
else
  puts ("x is equal to y");
```

3.3.10.3 The Less-Than Operator

Use the less-than operator `<` to determine if the first operand is less than the second operand. If it is, the result is 1. Otherwise, the result is 0.

```
if (x < y)
  puts ("x is less than y");
else
  puts ("x is not less than y");
```

3.3.10.4 The Less-Than-or-Equal-to Operator

Use the less-than-or-equal-to operator `<=` to determine if the first operand is less than or equal to the second operand. If it is, the result is 1. Otherwise, the result is 0.

```
if (x <= y)
  puts ("x is less than or equal to y");
else
  puts ("x is not less than or equal to y");
```

3.3.10.5 The Greater-Than Operator

Use the greater-than operator `>` to determine if the first operand is greater than the second operand. If it is, the result is 1. Otherwise, the result is 0.

```
if (x > y)
  puts ("x is greater than y");
else
  puts ("x is not greater than y");
```

3.3.10.6 The Greater-Than-or-Equal-to Operator

Use the greater-than-or-equal-to operator `>=` to determine if the first operand is greater than or equal to the second operand. If it is, the result is 1. Otherwise, the result is 0.

```
if (x >= y)
  puts ("x is greater than or equal to y");
else
  puts ("x is not greater than or equal to y");
```

3.3.11 Logical Operators

You can use the logical operators to test the truth value of two operands. The operands can be expressions of a primitive type, or pointers.

Note that while the comparison operators return the value 1 for a true expression, any nonzero expression is considered true in C.

3.3.11.1 The Logical AND Operator

Use the logical AND operator `&&` to test if two expressions are both true. If the first one is false, then the second one is not evaluated.

```
if ((x == 5) && (y == 10))
  printf ("x is 5 and y is 10");
```

You can also build an expression using more than one AND operator, and more than two operands, like this:

```
if ((x == 5) && (y == 10) && (z == 15))
  printf ("x is 5 and y is 10 and z is 15");
```

3.3.11.2 The Logical OR Operator

Use the logical OR operator `||` to test if at least one of two expressions is true. If the first expression is true, then the second expression is not evaluated.

```
if ((x == 5) || (y == 10))
   printf ("x is 5 or y is 10");
```

You can also build an expression using more than one OR operator, and more than two operands, like this:

```
if ((x == 5) || (y == 10) || (z == 15))
    printf ("x is 5 or y is 10 or z is 15");
```

3.3.12 Assignment Operators

You use the assignment operators to give values to variables. The first operand—the operand to which a value is being assigned, also known as the "lvalue"—cannot be a literal value or any other constant value. Except as noted, the operands must be of a primitive data type, or a pointer.

3.3.12.1 The Assignment Operator

Use the standard assignment operator `=` to assign the value of its right operand to its left operand. Unlike the other assignment operators, you can use this operator with variables of a structure type, in addition to primitive types and pointers.

```
x = 10;
y = 45 + 2;
z = (2 * (3 + function () ));
```

3.3.12.2 The Compound Assignment Operators

You use the compound assignment operators to perform an operation on both the left and right operands, and then assign the resulting expression to the left operand. Here is a list of the compound assignment operators, and a brief description of what they do:

- `+=`

  This operator adds its two operands together, and then assigns the result of the addition to the left operand.
- `-=`

  This operator subtracts its right operand from its left operand, and assigns the result of the subtraction to the left operand.

- `*=`

  This operator multiplies its two operands together, and then assigns the result of the multiplication to the left operand.
- `/=`

  This operator divides its left operand by its right operand, and assigns the result of the division to the left operand.
- `%=`

  This operator performs modular division on its operands, and assigns the result of the division to the left operand.
- `<<=`

  This operator performs a left shift operation on its left operand, shifting by the number of bits specified by the right operand, and assigns the result of the shift to the left operand.
- `>>=`

  This operator performs a right shift operation on its left operand, shifting by the number of bits specified by the right operand, and assigns the result of the shift to the left operand.
- `&=`

  This operator performs a bitwise AND operation on its two operands, and assigns the result of the operation to the left operand.
- `^=`

  This operator performs a bitwise exclusive OR operation on its two operands, and assigns the result of the operation to the left operand.
- `|=`

  This operator performs a bitwise inclusive OR operation on its two operands, and assigns the result of the operation to the left operand.

Here is an example of using one of the compound assignment operators:

```
x += y;
```

That produces the same result (since there no side effects in the simple variable `x` as an lvalue) as:

```
x = x + y;
```

3.3.13 The Comma Operator

You use the comma operator `,` to separate two expressions. For instance, the first expression might produce a value that is used by the second expression:

```
x++, y = x * x;
```

More commonly, the comma operator is used in `for` statements, like this:

```
/* Using the comma operator in a for statement. */

for (x = 1, y = 10;  x <=10 && y >=1;  x++, y--)
  {
    ...
  }
```

This lets you conveniently set, monitor, and modify multiple control expressions for the `for` statement.

A comma is also used to separate function parameters; however, this is *not* the comma operator in action. In fact, if the comma operator is used as we have discussed here in a function call (without enclosing it in an additional set of parentheses), then the compiler will interpret that as calling the function with an extra parameter.

```
function (x,  y=47, y,  z);
```

Even though what may be intended by such a function call is to call the function with the parameters `x`, `y`, and `z`, with `y` set to 47, what will happen is that the function will be called with the parameters `x`, `y=47`, `y`, and `z`. So if you want to include expressions that use the comma operator in a function call, surround the comma operator expression with parentheses.

```
function (x,  (y=47, y),  z);
```

That will call `function` with the parameters `x`, `y`, and `z`, with `y` set to 47.

3.3.14 Member Access Operators

You can use the member access operator `.` to access the members of a structure or union variable. You put the name of the structure variable on the left side of the operator, and the name of the member on the right side.

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

struct point first_point;

first_point.x = 0;
first_point.y = 5;
```

You can also access the members of a structure or union variable via a pointer by using the indirect member access operator `->`. `x->y` is equivalent to `(*x).y`.

```
struct fish
  {
    int length, weight;
  };

struct fish salmon;

struct fish *fish_pointer = &salmon;

fish_pointer->length = 3;
fish_pointer->weight = 9;
```

See Section 2.6 [Pointers], page 19.

3.4 The Ternary Operator

C has only one ternary operator — the conditional operator `? :`.

You use the conditional operator to cause the entire conditional expression to evaluate to either its second or its third operand, based on the truth value of its first operand.

Put the first operand is before the question mark; this operand may be any expression. If it evaluates to true (nonzero), then the second operand (which is also an expression), which is placed in between the question mark and the colon, is evaluated, and becomes the value of the conditional expression. Otherwise, the third operand (also an expression), which is placed after the colon, is evaluated, and becomes the value of the conditional expression. Here is an example

```
a = (x == 5) ? y : z;
```

In that example, if `x` equals 5, then `a` will receive the value `y`. Otherwise, `a` will receive the value `z`. This can be considered a shorthand method for writing a simple `if...else` statement. The following example will accomplish the same task as the previous one:

```
if (x != 0)
    a = y;
else
    a = z;
```

If the first operand of the conditional operator is true, then the third operand is never evaluated. Similarly, if the first operand is false, then the second operand is never evaluated. The first operand is always evaluated.

3.5 Order of Evaluation

When an expression consists of subexpressions, such as `a + b * f()`, the subexpressions are not all evaluated at once, nor are they necessarily evaluated left to right as they appear in the expression. In the preceding example, for instance, the function call `f()` will be evaluated first, the result of which will be multiplied by `b`, the result of which will be added to `a`. That is not intuitive simply by looking at the expression, but there is a definite order in which subexpressions are evaluated.

The following is a list of types of expressions, presented in the same order that they are evaluated. The expressions that are evaluated first are said to have the highest precedence, and those that are evaluated last the low precedence. Some types of expressions have the same precedence. If two or more subexpressions have the same precedence, then they are usually evaluated left to right. If they are *not* evaluated left to right, then that is explicitly stated in the list.

1. Function calls, array subscripting, and membership access operator expressions.
2. Logical negation, bitwise complement, increment, decrement, unary positive, unary negative, indirection operator, address operator, type casting, and `sizeof` expressions. When used as subexpressions, these are evaluated right to left.
3. Multiplication, division, and modular division expressions.
4. Addition and subtraction expressions.
5. Bitwise shifting expressions.

6. Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions.
7. Equal-to and not-equal-to expressions.
8. Bitwise AND expressions.
9. Bitwise exclusive OR expressions.
10. Bitwise inclusive OR expressions.
11. Logical AND expressions.
12. Logical OR expressions.
13. Conditional operator expressions (using `?:`). When used as subexpressions, these are evaluated right to left.
14. All assignment expressions, including compound assignment. When multiple assignment statements appear as subexpressions in a single larger expression, they are evaluated right to left.
15. Comma operator expressions.

4 Statements

You write statements to cause actions and to control flow within your programs. You can also write statements that do not do anything at all, or do things that are uselessly trivial.

4.1 Labels

You can use labels to identify a section of source code. A label consists of an identifier (such as those used for variable names) followed by a colon. Here is an example:

```
treet:
```

You should be aware that label names do not interfere with other identifier names:

```
int treet = 5;    /* treet the variable. */
treet:            /* treet the label. */
```

The ISO C standard mandates that a label must be followed by at least one statement (possibly a null statement). GCC will compile code that does not meet this requirement, but be aware that if you violate it, your code may have portability issues.

4.2 Expression Statements

You can turn any expression into a statement by adding a semicolon to the end of the expression. Here are some examples:

```
5;
2 + 2;
10 >= 9;
```

In each of those, all that happens is that each expression is evaluated. However, they are useless because they do not store a value anywhere, nor do they actually do anything, other than the evaluation itself. The compiler is free to ignore such statements.

Expression statements are only useful when they have some kind of side effect, such as storing a value, calling a function, or (this is esoteric) causing a fault in the program. Here are some more useful examples:

```
x++;
y = x + 25;
puts ("Hello, user!");
*cucumber;
```

The last of those statements, `*cucumber;`, could potentially cause a fault in the program if the value of `cucumber` is both not a valid pointer and has been declared as `volatile`.

4.3 The `if` Statement

You can use the `if` statement to conditionally execute part of your program, based on the truth value of a given expression. Here is the general form of an `if` statement:

```
if (test)
  then-statement
else
  else-statement
```

If *test* evaluates to true, then *then-statement* is executed and *else-statement* is not. If *test* evaluates to false, then *else-statement* is executed and *then-statement* is not. The `else` clause is optional.

Here is an actual example:

```
if (x == 10)
  puts ("x is 10");
```

If `x == 10` evaluates to true, then the statement `puts ("x is 10");` is executed. If `x == 10` evaluates to false, then the statement `puts ("x is 10");` is not executed.

Here is an example using `else`:

```
if (x == 10)
  puts ("x is 10");
else
  puts ("x is not 10");
```

You can use a series of `if` statements to test for multiple conditions:

```
if (x == 1)
  puts ("x is 1");
else if (x == 2)
  puts ("x is 2");
else if (x == 3)
  puts ("x is 3");
else
  puts ("x is something else");
```

4.4 The `switch` Statement

You can use the `switch` statement to compare one expression with others, and then execute a series of sub-statements based on the result of the comparisons. Here is the general form of a `switch` statement:

```
switch (test)
  {
    case compare-1:
      if-equal-statement-1
    case compare-2:
      if-equal-statement-2
    ...
    default:
      default-statement
  }
```

The `switch` statement compares *test* to each of the *compare* expressions, until it finds one that is equal to *test*. Then, the statements following the successful case are executed. All of the expressions compared must be of an integer type, and the *compare-N* expressions must be of a constant integer type (e.g., a literal integer or an expression built of literal integers).

Optionally, you can specify a default case. If *test* doesn't match any of the specific cases listed prior to the default case, then the statements for the default case are executed. Traditionally, the default case is put after the specific cases, but that isn't required.

```
switch (x)
  {
    case 0:
      puts ("x is 0");
      break;
    case 1:
      puts ("x is 1");
      break;
    default:
      puts ("x is something else");
      break;
  }
```

Notice the usage of the `break` statement in each of the cases. This is because, once a matching case is found, not only are its statements executed, but so are the statements for all following cases:

```
int x = 0;
switch (x)
  {
    case 0:
      puts ("x is 0");
    case 1:
      puts ("x is 1");
    default:
      puts ("x is something else");
  }
```

The output of that example is:

```
x is 0
x is 1
x is something else
```

This is often not desired. Including a `break` statement at the end of each case redirects program flow to after the `switch` statement.

4.5 The `while` Statement

The `while` statement is a loop statement with an exit test at the beginning of the loop. Here is the general form of the `while` statement:

```
while (test)
  statement
```

The `while` statement first evaluates *test*. If *test* evaluates to true, *statement* is executed, and then *test* is evaluated again. *statement* continues to execute repeatedly as long as *test* is true after each execution of *statement*.

This example prints the integers from zero through nine:

```
int counter = 0;
while (counter < 10)
  printf ("%d ", counter++);
```

4.6 The `do` Statement

The `do` statement is a loop statement with an exit test at the end of the loop. Here is the general form of the `do` statement:

```
do
  statement
while (test);
```

The `do` statement first executes *statement*. After that, it evaluates *test*. If *test* is true, then *statement* is executed again. *statement* continues to execute repeatedly as long as *test* is true after each execution of *statement*.

This example also prints the integers from zero through nine:

```
int x = 0;
do
  printf ("%d ", x++);
while (x < 10);
```

4.7 The `for` Statement

The `for` statement is a loop statement whose structure allows easy variable initialization, expression testing, and variable modification. It is very convenient for making counter-controlled loops. Here is the general form of the `for` statement:

```
for (initialize; test; step)
  statement
```

The `for` statement first evaluates the expression *initialize*. Then it evaluates the expression *test*. If *test* is false, then the loop ends and program control resumes after *statement*. Otherwise, if *test* is true, then *statement* is executed. Finally, *step* is evaluated, and the next iteration of the loop begins with evaluating *test* again.

Most often, *initialize* assigns values to one or more variables, which are generally used as counters, *test* compares those variables to a predefined expression, and *step* modifies those variables' values. Here is another example that prints the integers from zero through nine:

```
int x;
for (x = 0; x < 10; x++)
  printf ("%d ", x);
```

First, it evaluates *initialize*, which assigns `x` the value 0. Then, as long as `x` is less than 10, the value of `x` is printed (in the body of the loop). Then `x` is incremented in the *step* clause and the test re-evaluated.

All three of the expressions in a `for` statement are optional, and any combination of the three is valid. Since the first expression is evaluated only once, it is perhaps the most commonly omitted expression. You could also write the above example as:

```
int x = 1;
for (; x <= 10; x++)
  printf ("%d ", x);
```

In this example, `x` receives its value prior to the beginning of the `for` statement.

If you leave out the *test* expression, then the `for` statement is an infinite loop (unless you put a `break` or `goto` statement somewhere in *statement*). This is like using `1` as *test*; it is never false.

This `for` statement starts printing numbers at 1 and then continues indefinitely, always printing `x` incremented by 1:

```
for (x = 1; ; x++)
  printf ("%d ", x);
```

If you leave out the *step* expression, then no progress is made toward completing the loop—at least not as is normally expected with a `for` statement.

This example prints the number 1 over and over, indefinitely:

```
for (x = 1; x <= 10;)
  printf ("%d ", x);
```

You can use the comma operator (see Section 3.3.13 [The Comma Operator], page 33) for monitoring and modifying multiple variables in a `for` statement:

```
for (x = 1, y = 10; x <= 10, y >= 1; x++, y--)
  printf ("%d %d ", x, y);
```

This example assigns values to both `x` and `y`, checks that `x` is less than or equal to 10 and that `y` is greater than or equal to 1, and increments `x` and decrements `y`.

4.8 Blocks

A *block* is a set of zero or more statements enclosed in braces. Often, a block is used as the body of an `if` statement or a loop statement, to group statements together.

```
for (x = 1; x <= 10; x++)
  {
    printf ("x is %d\n", x);

    if ((x % 2) == 0)
      printf ("%d is even\n", x);
    else
      printf ("%d is odd\n", x);
  }
```

You can also put blocks inside other blocks:

```
for (x = 1; x <= 10; x++)
  {
    if ((x % 2) == 0)
      {
        printf ("x is %d\n", x);
        printf ("%d is even\n", x);
      }
    else
      {
        printf ("x is %d\n", x);
        printf ("%d is odd\n", x);
      }
  }
```

You can declare variables inside a block; such variables are local to that block.

```
{
  int x = 5;
  printf ("%d\n", x);
}
printf ("%d\n", x);    /* Compilation error! x exists only
                          in the preceding block. */
```

4.9 The Null Statement

The *null statement* is merely a semicolon alone.

```
;
```

A null statement does not do anything. It does not store a value anywhere. It does not cause time to pass during the execution of your program.

Most often, a null statement is used as the body of a loop statement, or as one or more of the expressions in a `for` statement. Here is an example of a `for` statement that uses the null statement as the body of the loop (and also calculates the integer square root of `n`, just for fun):

```
for (i = 1; i*i < n; i++)
  ;
```

Here is another example that uses the null statement as the body of a `for` loop and also produces output:

```
for (x = 1; x <= 5; printf ("x is now %d\n", x), x++)
  ;
```

4.10 The goto Statement

You can use the `goto` statement to unconditionally jump to a different place in the program. Here is the general form of a `goto` statement:

```
goto label;
```

You have to specify a label to jump to; when the `goto` statement is executed, program control jumps to that label. See Section 4.1 [Labels], page 37. Here is an example:

```
goto end_of_program;
...
end_of_program:
```

The label can be anywhere in the same function as the `goto` statement that jumps to it, but a `goto` statement cannot jump to a label in a different function.

You *can* use `goto` statements to simulate loop statements, but we do not recommend it—it makes the program harder to read, and GCC cannot optimize it as well. You should use `for`, `while`, and `do` statements instead of `goto` statements, when possible.

4.11 The break Statement

You can use the `break` statement to terminate a `while`, `do`, `for`, or `switch` statement. Here is an example:

```
int x;
for (x = 1; x <= 10; x++)
  {
    if (x == 8)
      break;
    else
      printf ("%d ", x);
  }
```

That example prints numbers from 1 to 7. When `x` is incremented to 8, `x == 8` is true, so the `break` statement is executed, terminating the `for` loop prematurely.

If you put a `break` statement inside of a loop or `switch` statement which itself is inside of a loop or `switch` statement, the `break` only terminates the innermost loop or `switch` statement.

4.12 The `continue` Statement

You can use the `continue` statement in loops to terminate an iteration of the loop and begin the next iteration. Here is an example:

```
for (x = 0; x < 100; x++)
  {
    if (x % 2 == 0)
      continue;
    else
      sum_of_odd_numbers + = x;
  }
```

If you put a `continue` statement inside a loop which itself is inside a loop, then it affects only the innermost loop.

4.13 The `return` Statement

You can use the `return` statement to end the execution of a function and return program control to the function that called it. Here is the general form of the `return` statement:

```
return return-value;
```

return-value is an optional expression to return. If the function's return type is `void`, then it is invalid to return an expression. You can, however, use the `return` statement without a return value.

If the function's return type is not the same as the type of *return-value*, and automatic type conversion cannot be performed, then returning *return-value* is invalid.

If the function's return type is not `void` and no return value is specified, then the `return` statement is valid unless the function is called in a context that requires a return value. For example:

```
x = cosine (y);
```

In that case, the function `cosine` was called in a context that required a return value, so the value could be assigned to `x`.

Even in contexts where a return value is not required, it is a bad idea for a non-`void` function to omit the return value. With GCC, you can use the command line option `-Wreturn-type` to issue a warning if you omit the return value in such functions.

Here are some examples of using the `return` statement, in both a `void` and non-`void` function:

```
void
print_plus_five (int x)
{
  printf ("%d ", x + 5);
  return;
}

int
square_value (int x)
{
  return x * x;
}
```

4.14 The `typedef` Statement

You can use the `typedef` statement to create new names for data types. Here is the general form of the `typedef` statement:

```
typedef old-type-name new-type-name
```

old-type-name is the existing name for the type, and may consist of more than one token (e.g., `unsigned long int`). *new-type-name* is the resulting new name for the type, and must be a single identifier. Creating this new name for the type does not cause the old name to cease to exist. Here are some examples:

```
typedef unsigned char byte_type;
typedef double real_number_type;
```

In the case of custom data types, you can use `typedef` to make a new name for the type while defining the type:

```
typedef struct fish
{
  float weight;
  float length;
  float probability_of_being_caught;
} fish_type;
```

To make a type definition of an array, you first provide the type of the element, and then establish the number of elements at the end of the type definition:

```
typedef char array_of_bytes [5];
array_of_bytes five_bytes = {0, 1, 2, 3, 4};
```

When selecting names for types, you should avoid ending your type names with a `_t` suffix. The compiler will allow you to do this, but the POSIX standard reserves use of the `_t` suffix for standard library type names.

5 Functions

You can write functions to separate parts of your program into distinct subprocedures. To write a function, you must at least create a function definition. It is a good idea also to have an explicit function declaration; you don't have to, but if you leave it out, then the default implicit declaration might not match the function itself, and you will get some compile-time warnings.

Every program requires at least one function, called `main`. That is where the program's execution begins.

5.1 Function Declarations

You write a function declaration to specify the name of a function, a list of parameters, and the function's return type. A function declaration ends with a semicolon. Here is the general form:

```
return-type function-name (parameter-list);
```

return-type indicates the data type of the value returned by the function. You can declare a function that doesn't return anything by using the return type `void`.

function-name can be any valid identifier (see Section 1.1 [Identifiers], page 2).

parameter-list consists of zero or more parameters, separated by commas. A typical parameter consists of a data type and an optional name for the parameter. You can also declare a function that has a variable number of parameters (see Section 5.5 [Variable Length Parameter Lists], page 48), or no parameters using `void`. Leaving out *parameter-list* entirely also indicates no parameters, but it is better to specify it explicitly with `void`.

Here is an example of a function declaration with two parameters:

```
int foo (int, double);
```

If you include a name for a parameter, the name immediately follows the data type, like this:

```
int foo (int x, double y);
```

The parameter names can be any identifier (see Section 1.1 [Identifiers], page 2), and if you have more than one parameter, you can't use the same name more than once within a single declaration. The parameter names in the declaration need not match the names in the definition.

You should write the function declaration above the first use of the function. You can put it in a header file and use the `#include` directive to include that function declaration in any source code files that use the function.

5.2 Function Definitions

You write a function definition to specify what a function actually does. A function definition consists of information regarding the function's name, return type, and types and names of parameters, along with the body of the function. The function body is a series of statements enclosed in braces. Here is the general form of a function definition:

```
return-type
function-name (parameter-list)
{
  function-body
}
```

return-type and *function-name* are the same as what you use in the function declaration (see Section 5.1 [Function Declarations], page 45).

parameter-list is the same as the parameter list used in the function declaration (see Section 5.1 [Function Declarations], page 45), except you *must* include names for the parameters in a function definition.

Here is an simple example of a function definition—it takes two integers as its parameters and returns the sum of them as its return value:

```
int
add_values (int x, int y)
{
  return x + y;
}
```

For compatibility with the original design of C, you can also specify the type of the function parameters *after* the closing parenthesis of the parameter list, like this:

```
int
add_values (x, y)
    int x, int y;
{
  return x + y;
}
```

However, we strongly discourage this style of coding; it can cause subtle problems with type casting, among other problems.

5.3 Calling Functions

You can call a function by using its name and supplying any needed parameters. Here is the general form of a function call:

```
function-name (parameters)
```

A function call can make up an entire statement, or it can be used as a subexpression. Here is an example of a standalone function call:

```
foo (5);
```

In that example, the function '`foo`' is called with the parameter `5`.

Here is an example of a function call used as a subexpression:

```
a = square (5);
```

Supposing that the function '`square`' squares its parameter, the above example assigns the value 25 to `a`.

If a parameter takes more than one argument, you separate parameters with commas:

```
a = quux (5, 10);
```

5.4 Function Parameters

Function parameters can be any expression—a literal value, a value stored in variable, an address in memory, or a more complex expression built by combining these.

Within the function body, the parameter is a local copy of the value passed into the function; you cannot change the value passed in by changing the local copy.

```
int x = 23;
foo (x);
...
/* Definition for function foo. */
int foo (int a)
{
  a = 2 * a;
  return a;
}
```

In that example, even though the parameter `a` is modified in the function '`foo`', the variable `x` that is passed to the function does not change. If you wish to use the function to change the original value of `x`, then you would have to incorporate the function call into an assignment statement:

```
x = foo (x);
```

If the value that you pass to a function is a memory address, then you can access (and change) the data stored at the memory address. This achieves a similar effect as pass-by-reference function parameters in other languages, but is not the same: the memory address is simply a value, just like any other value, and cannot itself be changed. The difference between passing a pointer and passing an integer lies in what you can do using the value within the function.

Here is an example of calling a function with a memory address parameter:

```
void
foo (int *x)
{
  *x = *x + 42;
}
...
int a = 15;
foo (&a);
```

The formal parameter for the function is of type pointer-to-`int`, and we call the function by passing it the address of a variable of type `int`. By dereferencing the pointer within the function body, we can both see and change the value stored in the address. The above changes the value of `a` to '`57`'.

Even if you don't want to change the value stored in the address, passing the address of a variable rather than the variable itself can be useful if the variable type is large and you need to conserve memory space. For example:

```
struct foo
{
  int x;
  float y;
  double z;
};

void bar (struct foo *a);
```

In this case, unless you are working on a computer with very large memory addresses, it will take less memory to pass a pointer to the structure than to pass an instance of the structure.

One type of parameter that is always passed as a memory address is any sort of array:

```
void foo (int a[]);
...
int x[100];
foo (x);
```

In this example, calling the function `foo` with the parameter `a` does not copy the entire array into a new local parameter within `foo`; rather, it passes `x` as a pointer to the first element in `x`. Be careful, though: within the function, you cannot use `sizeof` to determine the size of the array `x`—`sizeof` instead tells you the size of the pointer `x`. Indeed, the above code is equivalent to:

```
void foo (int *a);
...
int x[100];
foo (x);
```

5.5 Variable Length Parameter Lists

You can write a function that takes a variable number of arguments. To do this, the function needs to have at least one parameter of a known data type, but the remaining parameters are optional, and can vary in both quantity and data type.

You list the first parameter as normal, but then as the second parameter, use an ellipsis: '`...`'. Here is an example function prototype:

```
int add_multiple_values (int number, ...);
```

To work with the optional parameters in the function definition, you need to use macro functions that are defined in the library header file '`stdarg.h`', so you must `#include` that file. Those functions are described in a separate manual, but here is an example of using them:

```
int
add_multiple_values (int number, ...)
{
  int counter, total = 0;

  /* Declare a variable of type 'va_list'. */
  va_list parameters;
```

```
    /* Call the 'va_start' function. */
    va_start (parameters, number);

    for (counter = 0; counter < number; counter++)
      {
        /* Get the values of the optional parameters. */
        total += va_arg (parameters, int);
      }

    /* End use of the 'parameters' variable. */
    va_end (parameters);

    return total;
  }
```

To use optional parameters, you need to have a way to know how many there are. This can vary, so it can't be hard-coded, but if you don't know how many optional parameters you have, then you could have difficulty knowing when to stop using the 'va_arg' function. In the above example, the first parameter to the 'add_multiple_values' function, 'number', is the number of optional parameters actually passed. So, we might call the function like this:

```
sum = add_multiple_values (3, 12, 34, 190);
```

The first parameter indicates how many optional parameters follow it.

Also, note that you don't actually need to use 'va_end' function. In fact, with GCC it doesn't do anything at all. However, you might want to include it to maximize compatibility with other compilers.

See section "Variadic Functions" in *The GNU C Library Reference Manual*.

5.6 The `main` Function

Every program requires at least one function, called '`main`'. This is where the program begins executing. You do not need to write a declaration or prototype for `main`, but you do need to define it.

The return type for `main` is always `int`. You do not have to specify the return type for `main`, but you can. However, you *cannot* specify that it has a return type other than `int`.

In general, the return value from `main` indicates the program's *exit status*. Usually a value of zero indicates success and any other value indicates an error, but this is not guaranteed. You can `#include <stdlib.h>` and use the symbolic names `EXIT_SUCCESS` and `EXIT_FAILURE` if portability is a concern.

You can write your `main` function to have no parameters, or to accept parameters from the command line. To have no parameters, it is best to specify `void` as the parameter list.

Here is a very simple `main` function with no parameters:

```
int
main (void)
{
  puts ("Hi there!");
  return 0;
}
```

To accept command line parameters, you need to have two parameters in the `main` function, `int argc` followed by `char *argv[]`. You can change the names of those parameters, but they must have those data types—`int` and array of pointers to `char`. `argc` is the number of command line parameters, including the name of the program itself. `argv` is an array of the parameters, as character strings. `argv[0]`, the first element in the array, is the name of the program as typed at the command line; any following array elements are the parameters that followed the name of the program.

Here is an example `main` function that accepts command line parameters, and prints out what those parameters are:

```
int
main (int argc, char *argv[])
{
  int counter;

  for (counter = 0; counter < argc; counter++)
    printf ("%s\n", argv[counter]);

  return 0;
}
```

5.7 Recursive Functions

You can write a function that is recursive—a function that calls itself. Here is an example that computes the factorial of an integer:

```
int
factorial (int x)
{
  if (x < 1)
    return x;
  else
    return (x * factorial (x - 1));
}
```

Be careful that you do not write a function that is infinitely recursive. In the above example, once `x` is 1, the recursion stops. However, in the following example, the recursion does not stop until the program is interrupted or runs out of memory:

```
int
watermelon (int x)
{
  return (watermelon (x));
}
```

5.8 Static Functions

You can define a function to be static if you want it to be callable only within the source file where it is defined:

```
static int
foo (int x)
{
  return x + 42;
}
```

This is useful if you are building a reusable library of functions and need to include some subroutines that should not be callable by the end user.

6 Program Structure and Scope

Now that we have seen all of the fundamental elements of C programs, it's time to look at the big picture.

6.1 Program Structure

A C program may exist entirely within a single source file, but more commonly, any non-trivial program will consist of several custom header files and source files, and will also include and link with files from existing libraries.

By convention, header files (with a ".h" extension) contain variable and function declarations, and source files (with a ".c" extension) contain the corresponding definitions. Source files may also store declarations, if these declarations are not for objects which need to be seen by other files. However, header files almost certainly should not contain any definitions.

For example, if you write a function that computes square roots, and you wanted this function to be accessible to files other than where you define the function, then you would put the function declaration into a header file (with a ".h" file extension):

```
/* sqrt.h */

double
computeSqrt (double x);
```

This header file could be included by other source files which need to use your function, but do not need to know how it was implemented.

The implementation of the function would then go into a corresponding source file (with a ".c" file extension):

```
/* sqrt.c */
#include "sqrt.h"

double
computeSqrt (double x)
{
  double result;
  ...
  return result;
}
```

6.2 Scope

Scope refers to what parts of the program can "see" a declared object. A declared object can be visible only within a particular function, or within a particular file, or may be visible to an entire set of files by way of including header files and using `extern` declarations.

Unless explicitly stated otherwise, declarations made at the top-level of a file (i.e., not within a function) are visible to the entire file, including from within functions, but are not visible outside of the file.

Declarations made within functions are visible only within those functions.

A declaration is not visible to declarations that came before it; for example:

```
int x = 5;
int y = x + 10;
```

will work, but:

```
int x = y + 10;
int y = 5;
```

will not.

See Section 2.9 [Storage Class Specifiers], page 22, for more information on changing the scope of declared objects.

7 A Sample Program

To conclude our description of C, here is a complete program written in C, consisting of both a C source file and a header file. This program is an expanded version of the quintessential "hello world" program, and serves as an example of how to format and structure C code for use in programs for FSF Project GNU. (You can always download the most recent version of this program, including sample makefiles and other examples of how to produce GNU software, from `http://www.gnu.org/software/hello`.)

This program uses features of the preprocessor; for a description of preprocessor macros, see *The C Preprocessor*, available as part of the GCC documentation.

7.1 hello.c

```
/* hello.c -- print a greeting message and exit.

   Copyright (C) 1992, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002,
   2005, 2006, 2007 Free Software Foundation, Inc.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 3, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software Foundation,
   Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.  */

#include <config.h>
#include "system.h"

/* String containing name the program is called with.  */
const char *program_name;

static const struct option longopts[] =
{
  { "greeting", required_argument, NULL, 'g' },
  { "help", no_argument, NULL, 'h' },
  { "next-generation", no_argument, NULL, 'n' },
  { "traditional", no_argument, NULL, 't' },
  { "version", no_argument, NULL, 'v' },
  { NULL, 0, NULL, 0 }
};

static void print_help (void);
static void print_version (void);

int
main (int argc, char *argv[])
{
  int optc;
  int t = 0, n = 0, lose = 0;
```

```
  const char *greeting = NULL;

  program_name = argv[0];

  /* Set locale via LC_ALL.  */
  setlocale (LC_ALL, "");

#if ENABLE_NLS
  /* Set the text message domain.  */
  bindtextdomain (PACKAGE, LOCALEDIR);
  textdomain (PACKAGE);
#endif

  /* Even exiting has subtleties.  The /dev/full device on GNU/Linux
     can be used for testing whether writes are checked properly.  For
     instance, hello >/dev/full should exit unsuccessfully.  On exit,
     if any writes failed, change the exit status.  This is
     implemented in the Gnulib module "closeout".  */
  atexit (close_stdout);

  while ((optc = getopt_long (argc, argv, "g:hntv", longopts, NULL)) != -1)
    switch (optc)
      {
      /* One goal here is having --help and --version exit immediately,
         per GNU coding standards.  */
      case 'v':
        print_version ();
        exit (EXIT_SUCCESS);
        break;
      case 'g':
        greeting = optarg;
        break;
      case 'h':
        print_help ();
        exit (EXIT_SUCCESS);
        break;
      case 'n':
        n = 1;
        break;
      case 't':
        t = 1;
        break;
      default:
        lose = 1;
        break;
      }

  if (lose || optind < argc)
    {
      /* Print error message and exit.  */
      if (optind < argc)
        fprintf (stderr, _("%s: extra operand: %s\n"),
 program_name, argv[optind]);
      fprintf (stderr, _("Try `%s --help' for more information.\n"),
               program_name);
      exit (EXIT_FAILURE);
    }
```

```
  /* Print greeting message and exit. */
  if (t)
    printf (_("hello, world\n"));

  else if (n)
    /* TRANSLATORS: Use box drawing characters or other fancy stuff
       if your encoding (e.g., UTF-8) allows it.  If done so add the
       following note, please:

       [Note: For best viewing results use a UTF-8 locale, please.]
    */
printf (_("\
+---------------+\n\
| Hello, world! |\n\
+---------------+\n\
"));

  else
    {
      if (!greeting)
        greeting = _("Hello, world!");
      puts (greeting);
    }

  exit (EXIT_SUCCESS);
}



/* Print help info.  This long message is split into
   several pieces to help translators be able to align different
   blocks and identify the various pieces.  */

static void
print_help (void)
{
  /* TRANSLATORS: --help output 1 (synopsis)
     no-wrap */
        printf (_("\
Usage: %s [OPTION]...\n"), program_name);

  /* TRANSLATORS: --help output 2 (brief description)
     no-wrap */
  fputs (_("\
Print a friendly, customizable greeting.\n"), stdout);

  puts ("");
  /* TRANSLATORS: --help output 3: options 1/2
     no-wrap */
  fputs (_("\
  -h, --help          display this help and exit\n\
  -v, --version       display version information and exit\n"), stdout);

  puts ("");
  /* TRANSLATORS: --help output 4: options 2/2
     no-wrap */
  fputs (_("\
```

```
  -t, --traditional       use traditional greeting format\n\
  -n, --next-generation   use next-generation greeting format\n\
  -g, --greeting=TEXT     use TEXT as the greeting message\n"), stdout);

  printf ("\n");
  /* TRANSLATORS: --help output 5 (end)
     TRANSLATORS: the placeholder indicates the bug-reporting address
     for this application.  Please add _another line_ with the
     address for translation bugs.
     no-wrap */
  printf (_("\
Report bugs to <%s>.\n"), PACKAGE_BUGREPORT);
}



/* Print version and copyright information.  */

static void
print_version (void)
{
  printf ("hello (GNU %s) %s\n", PACKAGE, VERSION);
  /* xgettext: no-wrap */
  puts ("");

  /* It is important to separate the year from the rest of the message,
     as done here, to avoid having to retranslate the message when a new
     year comes around.  */
  printf (_("\
Copyright (C) %s Free Software Foundation, Inc.\n\
License GPLv3+: GNU GPL version 3 or later\
<http://gnu.org/licenses/gpl.html>\n\
This is free software: you are free to change and redistribute it.\n\
There is NO WARRANTY, to the extent permitted by law.\n"),
              "2007");
}
```

7.2 system.h

```
/* system.h: system-dependent declarations; include this first.
   Copyright (C) 1996, 2005, 2006, 2007 Free Software Foundation, Inc.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 3, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software Foundation,
   Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.  */

#ifndef HELLO_SYSTEM_H
```

```
#define HELLO_SYSTEM_H

/* Assume ANSI C89 headers are available.  */
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Use POSIX headers.  If they are not available, we use the substitute
   provided by gnulib.  */
#include <getopt.h>
#include <unistd.h>

/* Internationalization.  */
#include "gettext.h"
#define _(str) gettext (str)
#define N_(str) gettext_noop (str)

/* Check for errors on write.  */
#include "closeout.h"

#endif /* HELLO_SYSTEM_H */
```

GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

   The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

   Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

A

B

C

D

E

F