4

Functions

Procedures Called To Perform Work

4.1 Introducing Functions

When you run a program that has been compiled, all the instructions that tell the computer what to do are contained in functions. Look again at the program ***demo.c:***

```
/* My first C language program */
#include <stdio.h>
main()
{
    printf("Welcome to the C language\n");
}
```

Example 4.1: examples/part1/demo.c

Notice that the expression ***main***() is followed by a left curly brace **{**, and that the program ends with a right curly brace **}**. This program contains a single function called ***main***.

The function named ***main()*** is a very special one in C, because every program begins execution with a function with that name, and all other functions in the program are called from it. This special function can be envisioned as the trunk of a tree: other functions (branches) are called from it and others (smaller branches) are called from those functions, and so on. There is no practical limit to how far this sequence may grow.

The following program ***minprog.c*** in **Example 4.2** is the minimum legal C program. Go ahead. Take a second to enter, compile and run it. It compiles without error, and, when run, does absolutely nothing because there is nothing between the **{}** pair.

```
/* minprog.c: The minimum C program */

main()
{
}
```

Example 4.2: examples/part1/minprog.c

Just as a reminder, you compile and run this program like this:

```
% gcc miniprog.c
% ./a.out
```

The name given to a function —here ***main***()— like the name given a variable, identifies it. The rules for creating a legal function name are identical to those for creating a variable name:

- Function names must begin with a letter or an underscore character, (_).
- Function names may contain only letters, digits (0...9), and the underscore character (_).
- Uppercase letters are distinct from lowercase letters. (on most compilers)
- Function names may not be the same as any keywords. (**char** and **int** are keywords).
- Function names must be different from each other. On some very old compilers, only the first 8 or fewer characters are considered unique.

- Functions name must not be the same as variable names within the same scope. (We will cover scope later in this book.)

A function may not be declared (may not begin) within a “{}” pair. Thus:

```
main()
{
   sub()    ← You may not declare a function inside another function.
   {
   }
}
```

It is not legal to declare ***sub***() like this because it is enclosed by the curly brace pair of ***main***(). The correct way to declare both as functions is like this:

```
main()
{
}
sub()
{
}
```

In this case, ***sub***() becomes subroutine that can be called from ***main***(). Consider the brief program called ***callsub.c*** in **Example 4.3**:

```
/* callsub.c: illustrate a function call */
#include <stdio.h>
main()               /* main declared */
{
    printf("This is main\n");
    sub();     /* sub called from main  */
    printf("Back in main\n");
}
sub()                /* sub declared  */
{
    printf("and this is sub\n");
}
```

Example 4.3: examples/part1/callsub.c

The above program the process of calling another function from ***main***(). Note that the program's "flow" is from ***main***() to ***sub***(), and, when ***sub***() has nothing left to do, back to ***main***() again.

4.2 Function Arguments

Arguments are the means by which values are passed to a function. For example, ***printf***() is a function that prints its arguments to the video screen. When you call ***printf***() with the statement:

```
printf("hello\n");
```

you are passing to it the single argument "hello\n". It then prints that argument to your screen.

A function's arguments are contained within the pair of parentheses that follow the function's name. A function is said to have an empty argument list when the parenthesis pair following the name is empty:

`main()` ← ***empty argument list, expects no arguments***

The parentheses may not be omitted, but they may be empty.

When a function is declared, its arguments must be listed in the same order in which it expects to receive them. Thus:

`sub(arg1, arg2, arg3)` ← *expects three arguments in this order*

declares ***sub***() to be a function which expects three arguments, and expects them to arrive in the order (from left to right) `arg1` first, `arg2` second, and `arg3` last.

4.2.1 Pre-ANSI Function Arguments

In addition to listing its expected arguments, the type (such as **int**, **float** or **char**) of each of those arguments must be declared. That declaration must occur after the right parentheses) and before the beginning curly brace {, as:

```
sub(arg1, arg2, arg3)
```
          ← *declarations of argument types go here*
```
{
```

In the following example:

```
show(firstarg, secondarg)
     char secondarg;
     int  firstarg;
{
```

the function named ***show***() expects two arguments. The names of the arguments and the order in which they will be received are `firstarg` first and then `secondarg second`. The first is declared to be of type **int**, and the second is declared to be of type **char**. Notice that the type declarations need not be in the same order as the arguments listed within the parentheses, but should be, as a friendly gesture to those who might later read your code.

To call the function ***show***(), from ***main***() for example, state its name followed by two arguments in parenthesis:

```
show('A', 65);
```

The arguments must be listed in the same order that ***show***() expects to receive them. The type of each should also match, because passing and receiving go hand in hand. The entire function call must be terminated by a semicolon.

Enter, compile and run the following program, ***callfunct.c*** in **Example 4.4**. Note that when we call the function ***show***() we pass it two values, one **char** and one **int**, and that those values are passed using variables.

```
/* callfunct.c: call a function */
#include <stdio.h>
main()
{
        char ch;
        int  num;

        ch = 'A';
        num = 65;
        show(ch, num);

        ch = 65;
        num = 'A';
        show(ch, num);
}

show(letter, number)
char letter;
int  number;
{
        printf("Letter is '%c'.\n", letter);
        printf("Number is %d.\n\n", number);
}
```

Example 4.4: examples/part1/callfunct.c

This program again demonstrates that characters may be interpreted as integers whose values correspond to those of letters, digits, and punctuation marks. The character constant ‘A’ and the integer constant 65 both have a numerical value of 65.

4.2.2 Modern Function Arguments

Modern C compilers (ANSI compliant such as gcc) allow the listing of function arguments and the declaration of types to occur together between the parentheses:

```
show(char letter, int number)
{
```

Here, the combined list is both easier to understand, and less prone to error. Modify the definition of ***show***() in ***callfunct.c*** **Example 4.4** to use this modern form of function declaration. Compile it and see if your change produced any errors. If it did (that is if your compiler doesn't understand this modern form) consider updating your compiler to the latest version.

We will continue to use this modern version of function declarations throughout his book. That is, instead of using the old style of function argument declarations:

```
sub(number, letter)
int number;
char letter;
{
```

We will instead use the new style of function argument declarations:

```
sub(int number, char letter)
{
```

4.3 Pass By Value

When you pass an argument to a function you are not passing the variable itself. What is passed is a copy of the value of that variable. The called function can do anything it wants to that copy without affecting the original variable's value in any way.

The program ***passbyvalue.c*** in **Example 4.5** demonstrates this property:

```
/* passbyvalue.c -- Pass by value */
#include <stdio.h>
main()
{
    int num = 1;
    printf("main before sub num = %d\n", num);
    sub(num);
    printf("main after  sub num = %d\n", num);
}

sub(int num)
{
    printf("sub got num = %d\n", num);
```

```
        num = 5;
        printf("sub set num = %d\n", num);
    }
```

Example 4.5: examples/part1/passbyvalue.c

Compile and run this program. It demonstrates that the variable `num` in ***main***() is passed to ***sub***(). The ***sub***() function prints the value it received (`1`). The ***sub***() function then changes the value in `num` to `5` and prints that new value. Then back in ***main***() the value of `main()`'s `num` is printed which is still `1`.

What happened is this. When ***main***() calls ***sub***() with an argument of `num`, the compiler does not pass the variable `num` to ***sub***(), instead it takes the value that the variable `num` contained and passes that value. In this way, any argument passed to a called function cannot be changed by that called function. (Later in Part 2, when you learn advanced topic you will find that pointer can form the exception to this rule.)

Passing by value is handy, especially to the beginner. You cannot use a function to change anything inside another function accidently.

4.4 Local Variables

Enter, compile and run the following program, ***localdemo.c*** in **Example 4.6**. Observe that variables declared within the functions are local to each. The variable `arg` in ***modify***() is different from the variable with the same name `arg` in ***main***(). What happens to one, does not affect what happens to the other.

```
    /* localdemo.c: Local variable scope */
    #include <stdio.h>
    main()
    {
        int arg = 1;

        printf("In main, arg = %d.\n", arg);
        modify();
        printf("back in main, arg = %d.\n", arg);
    }

    /* modify(): its personal copy of arg. */
    modify()
    {
        int arg = 5;

        printf("in modify, arg = %d.\n", arg);
        arg /= 2;
        printf("modify made arg = %d.\n", arg);
    }
```

Example 4.6: examples/part1/localdemo.c

Note that when ***main***() calls ***modify***() as,

```
modify();
```

Nothing is passed to ***modify***(). Instead, the function ***modify***() defines its own variable with the same name as the variable used in ***main***(). Even though ***modify***() modifies the value in `arg`, the value in ***main***()'s `arg` is unaffected. That is, even though both functions use the same variable name, each function has it own copy of that variable and the two can never interact.

4.5 The return Keyword

A function can return a value to the function that called it. The mechanism for returning a value is provided by the new keyword **return**.

The **return** keyword can be used either of two ways. If you want a function to return to the function that called it (whether or not it is finished) simply use the **return** keyword followed by a semicolon:

```
return;
```

If you want the function to return a value, simply place that value in parentheses following the **return** keyword:

```
return (value);
```

Some compilers are picky, and will flag as an error:

```
return ();
```

This ought to be the same as:

```
return;
```

that is, returning without providing a return value, however you should avoid its use.

The parentheses enclosing the returned value are optional (on most modern compilers), so we will omit them for clarity. This also has the advantage of preventing **return** from looking like a function:

```
return (value);   ← Legal but can be confused with a function..
return value;     ← Clearly is not a function.
```

Enter, compile and run the somewhat longer program ***returndemo.c*** in Example 4.7.

```
/* returndemo.c: Approaches to return. */
#include <stdio.h> /* for printf() */
main()
{
        int number;

        number = get5();
        printf("get5 returned %d.\n", number);
        number = get6();
        printf("get6 returned %d.\n", number);
        printf("get7 returned %d.\n", get7());
}

get5()  /* a function to return a value of 5 */
{
        return 5;
}
get6()  /* a function to return a value of 6 */
{
        int val = 6;
        return val;
}
get7()  /* a function to return a value of 7 */
{
        int val;
        return val = 7;
}
```

Example 4.7: examples/part1/returndemo.c

Note in this program that the function ***main***() uses the values returned from ***get5***() and ***get6***() where each is assigned to the variable `number`. The value of `number` is then printed.

The use of ***get7***() is somewhat different. The ***printf***() function, like all functions, expects its arguments to be passed by value. All we have done with ***get7***() is to pass its returned value directly, without first assigning it to a variable.

This illustrates that the value returned by a function can be treated exactly as though the function call itself possesses that value.

The program ***returndemo.c*** also demonstrates alternative ways of using the return statement. The function ***get5***() returned the constant value `5`. The function ***get6***() returned the value of the variable `val`. This variable was assigned the value `6`, so the value returned is that of `val` or `6`.

The function ***get7***() is a bit more roundabout. The value of the assignment

```
val = 7
```

is returned. Almost everything in C yields a usable value. The value of an assignment statement is the value of the result of the assignment.

4.6 Main() Returns A Value

As you learned earlier, the function ***main***() is the first one run in any C program. But what you have not learned yet is that ***main***() returns a value too.

The ***main***() function always returns the type **int** and so you should always declared it to be of type **int**:

```
int
main()
```

But note that once you give it a type you must also insure that before it exits it returns an integer value. When ***main***() returns zero it is returning success to the process (usually your shell) that launched it. When ***main***() returns non-zero it is returning an indication of failure to the process that launched it.

```
int
main()
{
    ...some code here
    return 0;
}
```

Here, the `return 0` corresponded to the **int** type of ***main***(). After all, zero is an integer value. If your shell cared whether or not your program succeeded, it would accept a **return** value of zero as a sign of success. Recall the minimum C program:

```
/* minprog.c: The minimum C program */

main()
{
}
```

In **Example 4.8** below we rewrite that minimum program with the type and return value of ***main***():

```
/* minprog2.c: the minimum C program */

int
main()
{
    return 0;
}
```

Example 4.8: examples/part1/minprog2.c

For the rest of this chapter we will always **return** 0 to the calling environment. In the next chapter you will learn about **if** and how to conditionally return failure.

4.7 Compiling Multiple Files

Now that we have covered functions, we need to backtrack and cover compiling again. To illustrate why, consider writing a function that reads an integer from the keyboard. Such a function would be valuable to many programs. So rather than including that function as a part of every C program we can make it a separate file and combine it using the compiler.

Suppose your function for reading an integer from the keyboard were called ***getd***(), you could write a separate file called ***getd.c*** and put just that function into it.

`getd.c`	←	*contains the getd() function*
`myprog.c`	←	*contains main()*

To combine these two source files into one program, just run the compiler the way you did earlier, but instead of listing one .c file, instead list them both:

```
% gcc getd.c myprog.c
```

The resulting program is still called ***a.out***, but this time that program contains the code from both files. Go ahead and try it. First create a file called getdempty.c as shown in **Example 4.9**:

```
/* getdempty.c */
int
getd()
{
    return 0;
}
```

Example 4.9: examples/part1/getdempty.c

The ***getdempty.c*** file contains a ***getd***() function that does nothing and always returns 0. In the next chapter you will learn how to use code loops to create a real ***getd***() function that does something useful.

Next modify ***minprog2.c*** in **Example 4.8** to call ***getd***(). In **Example 4.10** below we show how ***main***() can return the value returned by ***getd***():

```
/* minprog3.c: Minimum to call a function */
int
main()
{
    return getd();
}
```

Example 4.10: examples/part1/minprog3.c

Now compile these two files into a single program:

```
% gcc getdempty.c minprog3.c
```

When you run the resulting program, nothing happens because the combined program does essentially nothing. If you wish to see the exit value return by ***main***() you can do one of the following:

```
% ./a.out; echo $status     ← The csh shell or the tcsh shell
0

$ ./a.out; echo $?          ← The bourne shell, the bash shell
0
```

Even though you don't specify it in your program, the C compiler always includes a copy of the Standard C library in your program. The ***print***() function you have been using is part of that silently included standard library.

4.8 Explaining printf()

The ***printf***() function is a powerful member of the standard C library that does a lot more than just print words to your screen. The "f" stands for “formatting”, and ***printf***() can be used to print values, sentences, tables, and more. When you call ***printf***(), you must give it two pieces of information. The first piece (the mandatory one) is called the “control string”. The second (optional depending on the control string) is a list of values.

```
   printf("control string" ,val1, val2, ...);
```

The control string must be enclosed in full quotation marks. With two exceptions, everything so enclosed will be printed. For example:

```
   printf("hi there");
```

will print the words “hi there” directly to your video screen.

The exceptions are the two special format control directives, one of which begins with a backslash character \, and the other a percent character **%**.

4.8.1 The Backslash printf() Directive: \

When placed inside the control string, the backslash “\” directive is used to imbed control characters into the printed text. Control characters are those which control output. They can be produced by holding down the key marked “control” (or ctrl) while pressing a letter key. The “return” and “tab” keys also produce control characters. The most common control characters are represented in C as “\letter”, as shown in this table:

Table 4.3: printf() backslash controls

Backslash Expression	Control Character	Does What
`\b`	`Ctrl-h`	Backspace a single character.
`\f`	`Ctrl-l`	Formfeed, or new page. Clears the screen.
`\n`	`Ctrl-m`	Carriage return, move cursor down one line. This is the most commonly used one in C.

Table 4.3: printf() backslash controls

Backslash Expression	Control Character	Does What
`\r`	`Ctrl-r`	Linefeed, moves cursor to left margin.
`\t`	`Ctrl-h`	Horizontal tab.
`\\`		Double the backslash to print a single one.
`\”`		Imbed a full quotation mark.

For example, the `\n` in:

```
printf("Try saying that with any other
language!\n");
```

is there to produce a new line. This has the same effect as typing:

```
Try saying that with any other language!
```

on your keyboard, then pressing the return or enter key.

Create and compile the following short program ***pfdemo1.c*** in **Example 4.11**. Try to determine what it will print before you actually run it.

```
/* pfdemo1.c: The backslash character. */
#include <stdio.h>
main()
{
    printf("name\taka\tage\n");
    printf("----\t---\t---\n");
    printf("Robert\t\"Bob\"\t35\n");
    printf("Fido\tnone\t2\n");
}
```

Example 4.11: examples/part1/pfdemo1.c

Play with this program. Take a moment to find out what happens when ***printf***() is asked to print “\z” where z is not one of those letters listed.

4.8.2 The Percent printf() Directive: %

When placed in the control string, the percent % directive tells printf() to print a value at that place in the text. The type of that value is specified by a letter immediately following the %. An unknown letter will either produce an error or some unexpected output.

Table 4.4: printf() % characters

Character	Will Print
`%d`	Print an integer value.
`%c`	Print a single character.
`%f`	Print a floating point value.
`%s`	Print a string (see Chapter XXXX)
`%o`	Print an octal value (see Chapter XXXX)
`%x`	Print a hexadecimal value (see Chapter XXXX)
`%u`	Print an unsigned integer value (see Chapter XXXX)
`%e`	Print a double precision floating point value (see Chapter XXXX)
`%%`	Print a literal percent character.

For each "%" directive appearing in the control string, a corresponding value ***must*** appear in the value list. For example, to insert the integer value 5 into your output, you could:

```
    printf("The number five looks like %d\n", 5);
```

which prints to your video screen:

```
   The number five looks like 5
```

There can be as many “%” directives as you like. Just be sure that there are enough values, that they are in the right order, and that they are the right types.

Table 4.5: Order of % values

Control	**Prints What**
`"%d%c", value1, value2`	Prints value1 as an integer. Then prints value2 as a character. With no space between them.

Compile ***pfdemo2.c*** in **Example 4.12**. Try to understand what will be printed before running it. Try changing things around just to see what happens. Review **Table 4.4** and try to understand what all the % operators do for ***printf***().

```
/* pfdemo2.c: Illustrate % in printf(). */
#include <stdio.h>
main()
{
    char  ch  = 'C';
    int   num = 2;
    float pi  = 3.14159265;

    printf("%%c prints a character as ");
    printf("'%c'.\n", ch);
    printf("%%d prints an integer value as ");
    printf("%d\n", num);
    printf("%%f prints a float value as ");
    printf("%f\n", pi);
    printf("And %%%% prints a %% as ");
    printf("%%.\n");
}
```

Example 4.12: examples/part1/pfdemo2.c

In addition to printing a type, the **%** directive can be used to specify a field. This means that when the **%** directive is followed by a number, that number tells ***printf***() how many spaces the printed value will occupy:

```
printf(":%5d:", 99);
```

will print:

```
:   99:
```

If the field is larger than the number of digits, they will be printed to the right side of the field. If the size of the field is negative, the digits will be printed to the left side of the field. If the size of the field is too small, it will be expanded to accommodate the digits. If the field specification begins with a zero, the field will be zero filled from the left.

In the program ***pfdemo3*** in **Example 4.13** below, again try to predict what this program will print before you compile and run it.

```
/* pfdemo3.c: Field control with printf()  */
#include <stdio.h>
main()
{
    int num = 12345;

    printf("to right    :%20d:\n",  num);
    printf("to left     :%-20d:\n", num);
    printf("truncated   :%3d:\n",   num);
    printf("zero filled :%020d:\n", num);
}
```

Example 4.13: examples/part1/pfdemo3.c

In addition to specifying the size of the field, the `%f` directive allows you to specify the number of digits of precision —the number of digits to the right of the decimal point in a floating point number. To specify digits of precision add a dot (.) to the field specification, then follow the dot with the number of digits of precision you require. For example, `%7.2f` will print a floating point number with two digits following the decimal point, within a field of size 7. Thus:

```
printf(":%7.2f:", 1.234);
```

will print:

```
:   1.23:
```

Similarly, `%.2f` will print two digits to the right of the decimal point, with no restriction on the size of the field. Enter the program ***pfdemo4.c*** in **Example 4.14** and observe what it prints when it is run.

```
/* pfdemo4.c: Precision with printf() */
#include <stdio.h>
main()
{
    float val = 123.456;

    printf("just %%f  :%f:\n",     val);
    printf("%%10f     :%10f:\n",   val);
    printf("%%10.2f   :%10.2f:\n", val);
    printf("%%.2f     :%.2f:\n",   val);
}
```

Example 4.14: examples/part1/pfdemo4.c

As you have seen, ***printf***() provides a general purpose way to print many things in many ways to your screen. Take some time to experiment. Discover for yourself just how elegant ***printf***() can be. If you want to learn more about ***printf***(), read the manual page on your computer:

```
% man -s3c printf          ← On Solaris machines.
% man 3 printf             ← On FreeBSD machines
% man -s3 printf           ← On linux and OS-X machines
```

4.9 Things To Try

- Write a function that prints ten v's composed of a backslash followed by a slash (\/).
- Write a function that prints a 7 and then calls itself.

- Write a program with three functions none of which are called by ***main***().
- Write a program that calls a function to print a triangle using any character. For example:

```
    x
   x x
  x   x
 xxxxxxx
```

- Write a program that calls a function that calls a function, e.g. ***main***() calls ***sub1***() and then ***sub1***() calls ***sub2***().