1 A Closer Look at Functions

As mentioned at the beginning of this course, the main building blocks a C program are functions. A function has a declaration line, and a body. The declaration line contains the return type of the function, the function name, and the argument list of the function. The body of the function is whatever is within the pair of braces following the function declaration.

What does the parameter list of a function do? It declares the types and names of parameters that are to be passed to that function when the function is called. So, if the declaration line of a function reads

```
double pow(double a, double b)
```

it means that you are expected to call the function `pow` with two arguments, which are both of type `double`. There are a few important points about parameters and parameter passing in C: First, the parameter list also counts as the declaration of the variables. You should not try to re-declare those veriables within the function body. Second, parameters are always passed *by value* in C. What does that mean? It means that, when you pass a value to a function, a copy of that value is made, and whatever the function does it will do a *copy* of the value passed. In other words, whatever happens to the parameters stays in the function, and it will not affect any variables used in calling the function.

2 Function Prototypes

In C, a function must have been known by the compiler before it can be called by another function. This is because the compiler must know the argument list and how to pass the arguments before it can generate actual code for the function call. So far, we have not met a situation where this is a problem; we could always get around the problem by placing the functions in proper order. However, consider this example:

```
void b(void)
{
  a();
}

void a(void)
{
  b();
}
```

Here, function `a` calls function `b` so, `b` has to be defined before `a`. However, `b` calls `a` too, so `a` has to be defined before `b`. Both requirements can not be satisfied simultaneously. So, we need another solution. And that solution is using function prototypes.

A function prototype is simply the function declaration line without a function body and ending with a semicolon. The example above can easily be fixed by using prototypes:

```
void a(void);
```

```
void b(void);

void b(void)
{
  a();
}

void a(void)
{
  b();
}
```

Generally speaking, you do not need to specify variable names in a prototype. However, it is strongly reccomended that you do, because more often than not variable names do convey some information to the person reading the source code. Just for the sake of example, the two function prototypes are exactly the same as far as the compiler is concerned:

```
double log (double base, double argument);
double log (double, double);
```

Obviously, the first form is much more informative than the second, and that is the form you are expected use. Also, it is customary to place all function prototypes at the top of the code together rather than spreading them out in the code. Also, it is conventional to put the `main()` function at the top of the file it is in. You will be expected to follow these conventions from now on.

3 The Bitwise Operators of C

You have already seen the logical operators of the C language. You were probably curious why the logical "and" and "or" operators used the signs && and || signs rather than just & and |. The reason is that the "single" signs are reserved for the bitwise operators.

You already know that all numbers in C are stored in binary form. (In fact, all numbers in all digital computers today are stored in binary form.) C has bitwise operators that act on corresponding bits of each operand. The operators are as follows:

Operation	Operator
Bitwise AND	&
Bitwise OR	|
Bitwise NOT	~
Bitwise XOR	^

These operators carry out the requested logic operation, but on individual bits of the numbers involved.

4 Shift Operators

There are two more operators that operate on bit patterns in C, and those are the two shift operators, << (left shift) and >> (right shift). What they do is shift the bit patterns of the integers they act on. The general use is as follows:

```
<expression-to-be-shifted> << <number-of-left-shifts>;
<expression-to-be-shifted> >> <number-of-right-shifts>;
```

When a number is shifted left, all bits move one place to the left. The leftmost bit is lost. The remaining space on the right is filled with a zero. In a left-shift operation, this is repeated the number of times specified by the second argument to the operator.

When a number is shifted right, all bits move one place to the right. The rightmost bit is lost. The space on the left is filled with a zero if the original number was positive, and it is filled with a 1 if the original number was negative.

Note that shifting a number to the left one place is the same as multiplying the number by two. Also, shifting a number to the right is the same as dividing the number by two. The reason these operators exist anyway is because due to the fact that for many processors, shift operations are much faster compared to multiplications.

5 A Closer Look at Variable Scopes

Knowing what the scope of your variables is quite important. By that we mean where you can actually refer to a variable, and how long that variable will keep its value. The general rule is as follows. Variable declarations can only be made at the beginning of a block of code, delimited by a pair of braces. Those variables are only valid within that block of code, and once execution leaves the block, the variables are said to *fall out of scope*, and they are destroyed together with any values they might have. Should execution enter that block of code again, the variables will be created again, but they will not retain their earlier values. Obviously, the same goes for function arguments. Arguments are only defined for the function, and their values are lost as soon as execution of that function ends.

There is another possibility. You can define variables outside of any function, stright in the source code. In that case, those variables are *global* variables. Global variables are created when program execution begins, and they never lose their values until program execution ends. Any function can refer to them, use their values or modify their values. In general, functions should be self-contained, and not refer to or modify any global data. In other words, global variables should be used only where necessary. Too many global variables spoil the code.

6 Exercise 1

Write a C program that reads an integer from the keyboard, and prints out its bit pattern on the screen, as four eight-bit blocks. So, for example, if the user enters 1535, the computer should print:

```
00000000 00000000 00000101 11111111
```

Make sure this works for negative numbers. You should write a separate function that takes an integer argument, and prints it in the format shown above. Use a function prototype to declare the function, and place `main()` above the function.

7 Exercise 2

A trick sometimes used in programming is to use individual bits in an integer as one-bit flags. Write a C program that flips the value of the requested bit of an integer variable, and displays the variable on the screen. The program should continue asking for bit number and flipping bits until the user enters a negative number for the bit position. Here is what a sample run should look like:

```
00000000 00000000 00000000 00000000
Which bit shall I flip? 0
00000000 00000000 00000000 00000001
Which bit shall I flip? 7
00000000 00000000 00000000 10000001
Which bit shall I flip? 31
10000000 00000000 00000000 10000001
Which bit shall I flip? 7
10000000 00000000 00000000 00000001
Which bit shall I flip? -1
```

For this, you should make use of the function you have written in the previous exercise.

8 My Solution for Exercise 1

```
#include <stdio.h>

void print_binary(int x);

int main(void)
{
  int a;

  printf("Enter a number:");
  scanf("%d", &a);

  print_binary(a);

  return 0;
}


void print_binary(int x)
{
  unsigned int mask;
  int bits_printed=0;

  for (mask = 1 << 31; mask > 0; mask >>= 1) {
    printf((x & mask) ? "1" : "0");

    bits_printed++;

    if (bits_printed == 8) {
      printf(" ");
      bits_printed = 0;
    }

  }

  printf("\n");

}
```