Programming with C

Terry Marris November 2010

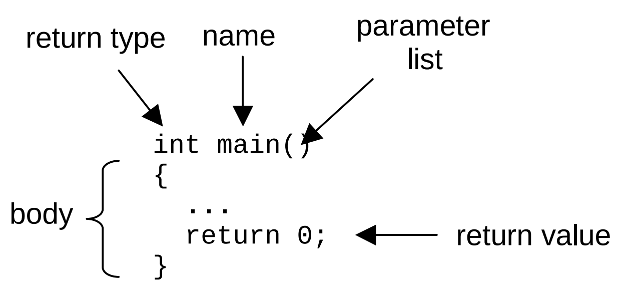
11 Functions

Previously we considered repetitions using the *for* statement. Now we move onto functions. Functions are the fundamental building block of all C programs.

11.1 Anatomy of a Function

We have used C library functions such as *printf()*, *gets()* and *atoi()*. We have completed the inbuilt C *main()* function. Now write some functions of our own.

A function has a name, a return type, a return value, a body, and a parameter list.



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

Our parameter list is empty. We look at parameter lists later. We choose function names that describe what the function does, or what the function returns. The return value must match the return type.

We write functions to perform useful tasks. Using functions saves us from having to repeat lines of code, allows us to re-use the same function in different places, and helps us manage program size and complexity. Up until now, our programs have been small and simple. But the time will come when you will be writing programs several thousand lines long.

11.2 Parameters and Arguments

When we use a function, we supply *argument* values. For example, in the call to *abs()* we supply an integer argument value.

```
int positiveNumber = abs(-3);
```

Here, the argument value is -3. You may remember *abs()* returns its argument without its minus sign, if it had one. (We looked at the *abs()* function in section 5.3.)

How might the *abs()* function itself be written?

```
/* abs: returns its parameter n, without its minus sign if it has one
*/
int abs(int n)
{
  if (n >= 0)
    return n;
  else                  parameter
    return n * -1;
}
```

Here, the parameter is the *int n* within the brackets of *int abs(int n)*.

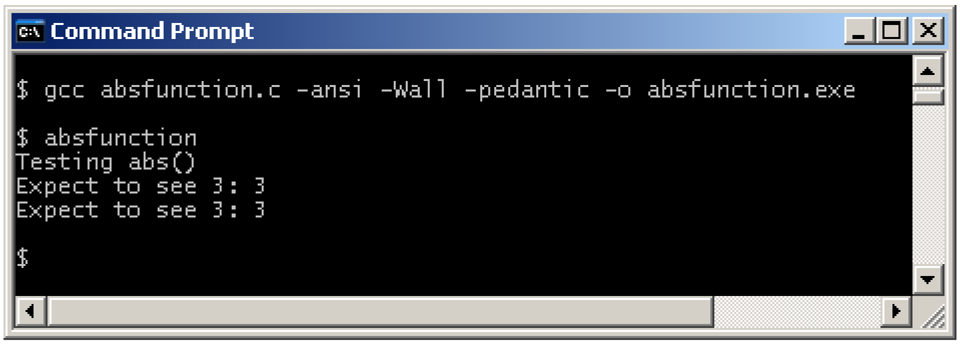
Arguments and parameters always occur in matched pairs. An argument is the value supplied to a function parameter. A parameter receives the argument value supplied. Of course, the argument value must match the parameter type.

In our *abs()* function, if the parameter is zero or more, it is returned unchanged. But if it is less than zero, it is multiplied by -1, thereby reversing its sign, and the result is returned.

We always include a comment describing *what* the function does. Look at the brackets that mark out the function's statement block: they are both written vertically in line with each other and in line with the function's return type. This is not compulsory, but it is a respected layout standard.

Here is the entire program and its run.

```
/* absfunction.c: implements and tests an absolute function */

#include <stdio.h>

/* abs: returns the argument n, without its minus sign if it has one
*/
int abs(int n)
{
  if (n >= 0)
    return n;
  else
    return n * -1;
}

int main()
{
  printf("Testing abs()\n");
  printf("Expect to see 3: %d\n", abs(3));
  printf("Expect to see 3: %d\n", abs(-3));
  return 0;
}
```



```
$ gcc absfunction.c -ansi -Wall -pedantic -o absfunction.exe

$ absfunction
Testing abs()
Expect to see 3: 3
Expect to see 3: 3

$
```

11.3 getString

A problem with using

```
char string[BUFSIZE];
gets(string);
```

to read a string entered at the keyboard is that *BUFSIZ* is huge and wastes a lot of memory. Can we reduce the space allowed for string input without the user entering more than we bargained for and wreaking havoc with our carefully constructed programs? We can have a try.

getchar() reads a single character entered at the keyboard. It is defined in *stdio.h*. We keep on reading characters entered, and storing them, until either the new line (return) key is pressed or we reach the specified maximum space allowed for the string. Any characters entered after the space has been filled are ignored. We ensure that the end of the string is marked with the null character since many string handling functions rely on it.

string size = 5

0	1	2	3	4
f	o	u	r	\0

With a size of 5, we can see that the value *four* just fits, but *forty* does not. The maximum length of the string is 4.

Here is our *getString()* function.

```
/* getString: reads a string from the keyboard, returns its length */
int getString(char string[], int maxLength)
{
  char c;
  int i = 0;

  while ((c = getchar()) != '\n') {
    if (i < maxLength - 1) {
      string[i] = c;
      i++;
    }
  }
  string[i] = '\0';
  return i;
}
```

The parameter *char string[]* says that *string* holds a numbered sequence of characters, how many characters is defined before the function is called - see *main()* below.

Look at

```
while ((c = getchar()) != '\n') {
```

Remember that the assignment operator has the lowest precedence of all the operators met so far. But we want it to be done first, and so we enclose it within brackets. Then we want to see whether the character entered is the new line character. We loop if it is not. Another way of writing this loop header is

```
for (c = getchar(); c != '\n'; c = getchar())
```

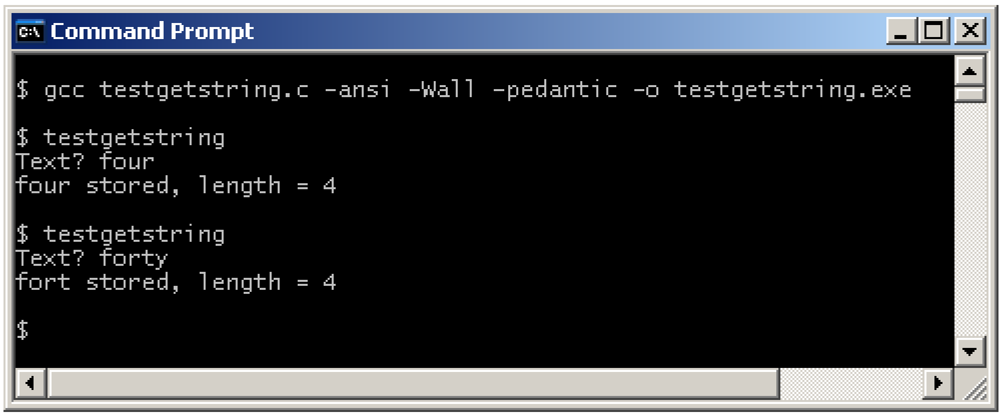
Get the first character. Loop if the character is not the new line character. Get the next character.

```
if (i < maxLength - 1) }
```

ensures that we never overfill the space reserved for the string no matter how many keys the user presses, and that we have space for the end-of-string null character, *\0*.

Here is the function in a test program, along with its run.

```
/* testgetstring.c: tests an implementation of getString() */

#include <stdio.h>

/* getString: reads a string from the keyboard, returns its length */
int getString(char string[], int maxLength)
{
  char c;
  int i = 0;

  while ((c = getchar()) != '\n') {
    if (i < maxLength - 1) {
      string[i] = c;
      i++;
    }
  }
  string[i] = '\0';
  return i;
}

int main()
{
  enum { size = 5 };

  char string[size];
  int length;

  printf("Text? ");
  length = getString(string, size);
  printf("%s stored, length = %d\n", string, length);
  return 0;
}
```



```
$ gcc testgetstring.c -ansi -Wall -pedantic -o testgetstring.exe

$ testgetstring
Text? four
four stored, length = 4

$ testgetstring
Text? forty
fort stored, length = 4

$
```

Ahh. What is this

```
enum { size = 5 };
```

Well, we use *size* twice within the test program, once in the declaration of *string* and once when we call *getString()*. We would like to have written *const int size = 5*, but declaring an array using this definition of *size* is not allowed in ANSI C. So instead we define *size* using an enumeration. An *enum* is just a list of constants, each of type *int*.

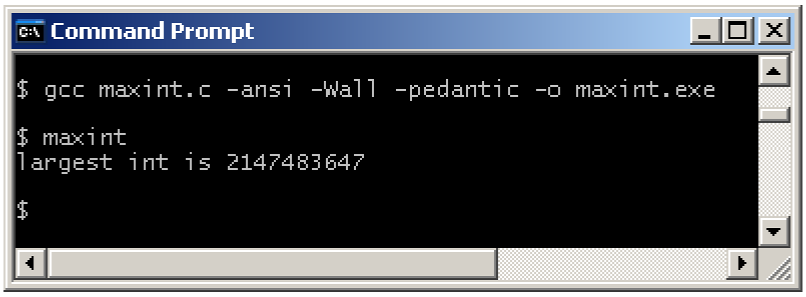
We can use the *getString()* function described above by copying and pasting it into our programs. Another way is to put it in our own library, and link the library object code with our programs. We do this in the next chapter.

11.4 getInt

We develop a function to read an integer input from the keyboard.

First, we determine the maximum size of an integer.

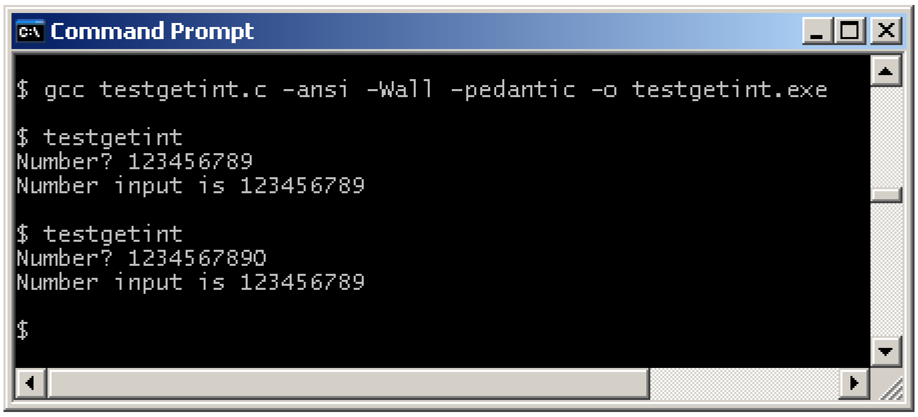
```
/* maxint.c: prints the largest integer value */

#include <stdio.h>
#include <limits.h>     /* defines INT_MAX */

int main()
{
  printf("largest int is %d\n", INT_MAX);
  return 0;
}
```



```
$ gcc maxint.c -ansi -Wall -pedantic -o maxint.exe

$ maxint
largest int is 2147483647

$
```

That size is 2 147 483 647 on my system. (On other systems that might be 32 767.) We see that there are 10 digits altogether. So, if we limit the input from the keyboard to nine digits, we should remain within the maximum bounds set for an integer.

```c
/* testgetint.c: tests an implementation of getInt() */

#include <stdio.h>
#include <stdlib.h>

/* getString: reads a string from the keyboard, returns its length */
int getString(char string[], int maxLength)
{
  char c;
  int i = 0;

  while ((c = getchar()) != '\n') {
    if (i < maxLength - 1) {
      string[i] = c;
      i++;
    }
  }
  string[i] = '\0';
  return i;
}


/* getInt: returns the integer entered at the keyboard */
int getInt()
{
  char string[10];

  getString(string, 10);  /* for 9 digits + end-of-string */
  return atoi(string);
}

int main()
{
  int n;

  printf("Number? ");
  n = getInt();
  printf("Number is %d\n", n);
  return 0;
}
```



```
$ gcc testgetint.c -ansi -Wall -pedantic -o testgetint.exe

$ testgetint
Number? 123456789
Number input is 123456789

$ testgetint
Number? 1234567890
Number input is 123456789

$
```

Here, we have tested the program twice, once with nine digits input, 123456789, once with 10. When 10 digits were entered the last digit was truncated (cut off). Perhaps we should display a warning when too many digits are entered, and ask the user to re-consider. That is a problem we intend to deal with in the chapter on validation. The important point is that the we have remained in control despite the user's input.

Incidentally, *atoi()* stops translating characters into an integer when it comes across the first non-digit character.

11.5 Order

The order in which functions appear in a program matters. A function must be *defined* or be *declared* before it is called so that the compiler can check whether the function has been called correctly.

A function definition is the whole, complete function implementation. A function declaration is just the function header. For example, a function definition is

```
/* getInt: returns the integer entered at the keyboard */
int getInt()
{
  char string[10];

  getString(string, 10);  /* for 9 digits + end-of-string */
  return atoi(string);
}
```

And a function declaration is

```
int getInt();
```

Notice the trailing semi-colon.

In general, we define functions before we use them. This means that *main()* is usually the last function listed in a program. We deal with function declarations in the next chapter.

Exercise 11.1

1. Write and test a function that returns the least of two integers.

2. Write and test a function that tests two numbers of type *double* for equality. The function should return true i.e. 1, if they are equal, false i.e. 0 otherwise.

3. The largest value of type double, *DBL_MAX*, is defined in *float.h*. Write program to print this largest value, and then write and test a function, *getDouble()*, that returns a value of type *double* entered at the keyboard.

We have seen how to write and test functions.

Next we see how to create a simple library of useful functions.

Bibliography

Kernighan B and Ritchie D *The C Programming Language* Prentice Hall 1988
Mark Williams Company *ANSI C A Lexical Guide* Prentice Hall 1988