Functions in C/C++

This background information document will provide you with a concise introduction to functions in C/C++, along with some useful miscellaneous topics.

1 Functions

By now, you know that it is easy to make mistakes when writing programs. The best-practice approach at minimising these 'bugs' is called *modular programming.* The key elements of this approach is to:

- Identify the structure of the problem, and split it into tasks.
- Write code for the tasks one at a time.
- Test each task independently.

We can use comments, indentations and judicious variables names in our source code to best reflect these separate sub-tasks, but by far the most useful programming concept for this purpose is that of *functions.* We have so far not made use of functions so our codes have been long sequences of commands[1]. If we continued this approach for more complex problems our codes would become very long and unwieldy. Moreover we would likely notice that many blocks of code have a similar structure. Using functions allows us to identify commonalities between tasks and extract sub-tasks as a self-contained reusable units. The end result is much more elegant and logical piece of code.

We have already made use of some functions, for example `sqrt()` from `<cmath>` library. This already illustrates that we can treat functions as **black boxes**. We don't need to know the precise implementation of a function. Rather we need a function specification that tells us (a) the type of input that is required/allowed, (b) what the function does with the input, and (c) what output (if any) is returned. So for `sqrt(x)` this is take a numerical input `x`, which could be `int`, `float` or `double`, compute its square-root, and return the result as a `double`. Once you have written a function, it should then be tested (debugged) thoroughly preferably in all contexts. A 'black box' is only useful insofar as it is reliable. Once you have a working function, you can build a hierarchy: write new functions that depend on the functions you've already written. This is how all software is developed.

1.1 Defining a function

The syntax for defining a C/C++ function is:

```
return_type <function_name>(arguments)
{
```

[1]Essentially we have been constructing compiled C/C++ 'scripts'.

```
        block of code
        return return_value;
    }
```

In fact every piece of code we have written so far has technically been a function following this definition:

```cpp
int main()
{
    // Our code was always put here

    return EXIT_SUCCESS;
}
```

Thus `main()` itself is a function with an integer return type, but it is a special mandatory function which is always executed first in any program. So far we have always defined `main()` as having no input arguments. We will see next week it can have some fixed input arguments corresponding to command line inputs. Here is a definition of a new function `square()` that calculates the square of a double:

```cpp
double square(double x)
{
    return x * x; // Do the calculation and return the value on the same line
}
```

Just as a function doesn't necessarily have to have an input it also doesn't have to have a return a value. In this case their return type is stated to be `void`, for example

```cpp
void display_message()
{
    std::cout << "This function does nothing but display this message!" << "\n";
    // No return statement is needed here
}
```

Here is a definition of another function `sum()` that returns the sum of two integers, so has two input arguments:

```cpp
int sum(int a, int b)
{
    int c = a + b; // Add up the inputs and assign to a local variable
    return c; // Return the value of the local variable
}
```

Notice that in `sum()` we have also defined a new variable `c` inside the function itself. We will discuss this more shortly. Obviously `square()` and `sum()` are very simple functions. However, we can define functions with any number of arguments of any type, including structures, and you can put as many lines of code as you want within the function. This allows functions to encapsulate highly non-trivial tasks. For readability though it is worth considering breaking large functions (more than 100's of lines of code) into a set of smaller, more digestible functions.

1.2 Calling a function

Having defined a function, as above, how do you go about using it in a program? You can call a function by using its name, and by passing arguments within round brackets ( ), e.g. `square(x)`. One possible form of a program illustrating this is:

```cpp
#include <iostream>

double square(double x)
{
    return x * x; // Here we just do the computation in the return line
}
```

```cpp

int main()
{
    double x, xsq;

    std::cin >> x; // Read in a value
    xsq = square(x); // Square its value using our function
    std::cout << x << " squared is " << xsq << "\n"; // Display answer

    return EXIT_SUCCESS;
}
```

Notice we put the function definition at the top of the source file outside `main()`. This is because the C/C++ compiler will read the code from the top of the file to the bottom. It needs to have seen the definition of `square()` before any calls to the function are made. However, sometimes in complex codes it can be difficult, or even impossible, to define a function before it is called because you may have many interdependent functions calling each other. To solve this problem, C/C++ allows you to declare a function separately from where it is defined. Declaring a function means specifying its name, arguments and return type, but no implementation code. We can rewrite our code as:

```cpp
#include <iostream>

double square(double x); // Declaration of the function

int main()
{
    double x, xsq;

    std::cin >> x; // Read in a value
    xsq = square(x); // Square its value using our function
    std::cout << x << " squared is " << xsq << "\n"; // Display answer

    return EXIT_SUCCESS;
}

double square(double x) // Definition of the function
{
    return x * x; // Our implementation
}
```

Note that the name, argument types and return types of the declaration and definition must all match perfectly otherwise the compiler will complain.

1.3 Scope of variables

A crucial part of functions is variable scope. Remember that in C/C++ any variables declared inside a block enclosed by `{ ... }` are defined only within the scope of that block. Specifically, they are local to that block and cannot be accessed outside of it. The same rule applies to functions. Any variables declared inside them are **local variables**, like `c` in `sum()` above, and are inaccessible outside the function. Unlike a block of code, which can access everything defined in the block outside of it, a function can only access what you pass to it as arguments, and the calling block of code will only see the returned value. This increased insulation makes functions very useful for avoiding accidental contamination in your code. As such different functions can have local variables with the same name because they are 'inside different black boxes'.

In C/C++ all variables passed as arguments to a function are **passed-by-value**. Suppose we rewrote our program using `square()` as:

```cpp
#include <iostream>

double square(double x); // Declaration of the function

```

```
 5 int main()
 6 {
 7     double x, xsq;
 8 
 9     std::cout << "Enter a value:" << std::endl;
10     std::cin >> x; // Read in a value
11     xsq = square(x); // Square its value using our function
12     std::cout << x << " squared is " << xsq << "\n"; // Display answer
13 
14     return EXIT_SUCCESS;
15 }
16 
17 double square(double z) // Definition of the function
18 {
19     double y = z * z; // Implement using a local variable
20     z = 2*y; // Now modify the input variable
21     return y;
22 }
```

A few changes have been made here. First, the implementation of `square()` calls its input `z` rather than `x` as in the declaration in line 3. This makes no difference since the declaration does not care about variables names, only their types[2]. The reason for doing this is so it is clearer that the input variable for `square()` is not the same variable as the one given as its argument in the function call on line 10. The variable `z` inside `square()` is a new variable local to this function whose value is initialised with the value of the variable `x` in `main()`. It should then be clear then that line 19 inside `square()` does not alter at all the value of the variable `x` in `main()`. Functions cannot alter the value of their inputs since the way we have defined them so far are **pass-by-value**. This could be an issue, for example if we passed a large `vector` object since its contents would get deep-copied every time we ran the function. Moreover, there will be occasions where we want functions that can alter their inputs. To do this we need to **pass-by-reference**.

1.4 Passing arguments by reference

We can alter our definition of functions so variable are passed by reference. Here is a simple example of a declaration for a function which swaps the value of two integers:

```
void swap_integers(int& a, int& b);
```

By inserting `&` in front of the arguments we are explicitly telling the compiler that `a` and `b` should be passed-by-reference. The function do not need a return value since it can modify the input variables directly. Here is a program implementing and using this function:

```
#include <iostream>

void swap_integers(int& a, int& b); // Declaration of the function

int main()
{
    int x = 67; // Declare and initialise two integers
    int y = 24;

    std::cout << "x = " << x << " and y = " << y << "\n"; // Display variables
    swap_integers(x, y); // Swap by passing variables themselves
    std::cout << "x = " << x << " and y = " << y << "\n";  // Display them again
    return EXIT_SUCCESS;
}

void swap_integers(int& a, int& b)  // Definition of the function
{
    int local_var; // A local variable is needed to perform a swap
    local_var = a;
```

[2]Indeed you can remove all the variable names from the declaration if you wish.

```
    a = b;
    b = local_var;
}
```

Notice that we can treat `a` and `b` in our implementation of the function as just normal local integer variables.

The examples above used primitive data types, but the same properties apply to structures and objects (see next week). Here is a simple example of a program that uses function which accepts a `vector` object and normalizes it:

```
#include <iostream>
#include <vector>
#include <cmath>

// Declaration functions:
double magnitude (std::vector<double> inputVector);  // Vector passed by value
double normalize (std::vector<double>& inputVector); // Vector passed by reference

int main()
{
    // Declare and initialise components of a vector:
    std::vector<double> myVector{-0.82,1.23,0.77,-13.21};

    double mag = normalize(myVector); // Normalize returns original magnitude
    // Display the magnitude of original vector:
    std::cout << "|myVector| = " << mag << std::endl;
    mag = magnitude(myVector); // Evaluate the magnitude again
    // Display the magnitude of the normalized vector:
    std::cout << "|normalize(myVector)| = " << mag << std::endl;
    return EXIT_SUCCESS;
}

// Definition of functions:
double magnitude(std::vector<double> inputVector)
{
    double result = 0.0;
    for (int i=0; i<inputVector.size(); i++)
    {
        result = result + inputVector[i] * inputVector[i];
    }
    return sqrt(result);
}

double normalize(std::vector<double>& inputVector)
{
    double mag = magnitude(inputVector);
    for (int i=0; i<inputVector.size(); i++)
    {
        inputVector[i]  = inputVector[i]/mag; // Overwrite the vector components
    }
    return mag; // Return the original magnitude
}
```

Note that in `magnitude()` we passed the input `vector` object by value and a deep-copy of `myVector` was made. In `normalize()` the vector `myVector` was passed by reference explicitly so the function has access to the original data and can modify it directly. In `main()` we confirm that its magnitude has indeed been made unity by `normalize()`.

1.5 Function overloading

There will be some occasions where we might have several different declarations of the same function. This is called *overloading* a function and is permitted so long as each declaration has a distinct set of arguments and data types allowing the compiler to identify which instance of a function is being used. Earlier we constructed the `sum()` defined for `int`'s. Clearly the same task may be need for `double`'s so in this program we overload the function:

```
#include <iostream>
```

```
 2
 3 int sum(int a, int b); // Declared with two int arguments
 4 double sum(double c, double d); // Declared with two double arguments
 5
 6 // Note that it is OK to declare the function twice, even using different argument names
 7 // (as long as the type are correct).
 8 double sum(double, double); // Indeed, you don't have to specify the argument names.
 9
10 int main()
11 {
12     std::cout << sum(10, 20) << std::endl;
13     std::cout << sum(3.14159, 2.71828) << std::endl;
14     return EXIT_SUCCESS;
15 }
16
17 // Definitions must use the same types as the declaration, and must provide argument
18 // names since they are referenced in the body of the function. Unlike the type,
19 // names don't have to match those used in the declarations (although it's good
20 // practice to keep them consistent when names are used.)
21
22 // Defined with two int arguments
23 int sum(int c, int d)
24 {
25     std::cout << "Sum of two ints is: ";
26     return c + d;
27 }
28
29 // Defined with two double arguments
30 double sum(double a, double b)
31 {
32     std::cout << "Sum of two doubles is: ";
33     return a + b;
34 }
```

If we changed line 13 to `sum(10.0, 20)` then the above code will not compile since we have not defined a version of the `sum()` function with mixed `double` and `int` arguments. □