Lesson 2 - Functions
Creating and Using Functions

Summary

We want to create two functions, one to add two numbers and one to subtract two numbers. These numbers are typed in by the user so we will need to pass these numbers to the functions. In addition, we want to get a return value so we can display the addition to the user before the program exits.

New Concepts

Functions, Forward Declarations and Parameters.

Functions

The `functions.cpp` listing shows a function that takes two numbers, adds them and then returns the total (line 5). Notice how the addition and the returning of the values occur in one statement. We could have split this up, but this is quicker to write and just as clear. Notice how we pass values to the function using the variables `x` and `y` (integers). These are known as parameters when discussing functions. In this function two parameters are passed of type integer. The `add` function simply performs addition of the parameters `x` and `y` before saving the result as an integer and returning the result to the calling code.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int add(int x, int y) {
6    return x + y;
7 }
8
9 int minus(int x, int y);
```

functions.cpp

On line 9 we see a `minus` function. But wait, the statements are missing! This is what is known as a forward declaration. We say what the function is so it can be used by code that occurs before the function is actually written. We do not actually say what the statements are until later. Remember, just like variables, functions are declared so the program knows about them.

On line 10 we begin the `main` function. Let's declare some variables and call our functions. Note on lines 13 to 17 we ask the user for some numeric values and save them in variables `a` and `b` respectively using `cout` and `cin`. Line 19 shows the syntax for calling our `add` function.

```
10 int main() {
11    int a, b, c, exit;
12
13    cout << "Please type in a number" << "\n";
14    cin >> a;
15
16    cout << "Please type in another number" << "\n";
17    cin >> b;
18
19     c = add(a, b);
20
21     cout << a << '+' << b << '=' << c << "\n";
22
23     cout << a << '+' << b << '=' << add(a, b) << "\n";
24
25     cout << "enter another number to exit" << "\n";
26
27     cin >> exit;
28
29     return 0;
30 }
31
32 int minus(int x, int y) {
33     return x - y;
34 }
```

functions.cpp

Note that we can rewrite the two lines of code on lines 19 and 21, to one line as shown on line 23. Finally, note on line 30 we must provide the function definition belonging to the forward declaration from line 9.

C++ Style

The source code presented in this lesson has been placed within a C++ source file called `functions.cpp` (as denoted by the suffix '.cpp'). It is good practise, however, as the complexity of your code grows to place function (and class) definitions in their own header file. A header file may end in either .h or .hpp but in both cases, the header provides instructions on how to invoke any functions that you may define.
Header file declarations of functions generally appear as in the format of the `minus` function forward declaration shown on Line 9, where the definition (lines 32-33) exists in a source file. You may also 'inline' functions in header files, however. Inlining involves placing the definition directly into the header file. When your program is compiled, inlined functions are copied directly into the source code. This can sometimes speed up the execution of your program because their is a small overhead to invoking a function, which inlining avoids. Beware though, as inlining has its drawbacks. For example, if your inlined function contains many lines of code and is called frequently, this can substantially increase the size of your compiled program.

New Feature

C++11 includes a powerful feature called the *Lambda*. A lambda expression behaves like an anonymous function (i.e. a function which a body but no name) which can access variables from an enclosing scope. One of the numerous benefits of lambda expressions is that they promote modularity of code and for a large software project (like a Game Engine), modularity is essential.

The code snippet below shows a simple demonstration of lambda expressions, including the saving of a lambda expression as a variable which can be passed to another function. In the `main` function, add and minus lambda expressions are defined (lines 42 and 45) and assigned to variables of type `std::function<>`. Within the angle brackets you specify the return type followed by the function arguments contained within round brackets (i.e. `(int, int)`). The lambda expression follows the equals sign and begins with a set of square brackets (`[]`). Although empty in this example, in the square brackets you can declare any variables from the enclosing scope you wish to pass to the lambda. Following the square brackets you define the function arguments and lastly the body of the lambda.

```
35 #include <functional>
36
37 int processEvents(std::function<int(int, int)> someEvent, int x, int y) {
38    return someEvent(x, y);
39 }
40
41 int main(int argc, char* argv[]) {
42    std::function<int(int, int)> onAddEvent = [](int x, int y) {
43       return x + y;
44    };
45    std::function<int(int, int)> onMinusEvent = [](int x, int y) {
46       return x - y;
47    };
48    std::cout <<  "added: " << processEvents(onAddEvent, 4, 5) << std::endl;
49    std::cout << "subtracted: " << processEvents(onMinusEvent, 4, 5) << std::endl;
```

functions.cpp

In the code provided we have passed our two lambda expressions (`onAddEvent` and `onMinusEvent`) to a `processEvents` function (lines 48 and 49), effectively passing a function as an argument to another function! The benefit of this approach is that the `processEvents` function doesn't need to know anything about the operation the lambda performs, only the function signature. As your software grows in complexity, this kind of decoupling will make maintaining and extending your code much easier!

Exercises

1. Not every function we may write has to return a value. Write a function called `product` which accepts two integers as arguments (like `add` and `minus`) but returns no value. In the function body the `product` function should use `cout` to print the product directly to screen.

2. Write a function called `quotient` which should take a double argument and an integer argument. This function should also return a double data type. Now if you provide the `quotient` function with the values 5 and 3 say, it should return the value 1.66667 or thereabouts. Test this by calling the function within a `cout` statement.

3. Write a "calculator program" which asks for two numbers and a mathematical operator (represented as a `char` type. Depending on the operator (+, -, * or /) call the appropriate function and display the result. Use a `switch` statement to filter the choices.