IB9N7 C++ for Quantitative Finance

Lecture 6: Functions

J.F.Reizenstein@warwick.ac.uk

16 November 2015
(Week 7)

Outline

In this lecture, we'll cover the following topics:

- Functions.
  - Declaring, defining and calling.
  - Passing parameters by value and by reference.
- Header files etc.

Functions

Basic concepts

It is always a good idea when programming to split things up into easy conceptual chunks. In C++, this is achieved by

- Splitting code into functions.
- Placing data and functions into classes (we will discuss classes in the last week of this term).

The advantages of modularisation:

- Code becomes easier to manage.
- Easier to debug and test code.
- Enable easier re-use of code and allows reduced repetition.
  Recall that unnecessary duplication of code should be avoided.

Functions

Function

A function is a set of statements which together form a functional unit (e.g. perform a specific task) within the program. A function may have zero or more parameters (for accepting arguments) and may optionally return a value.

Best practice:

- One function should do one (conceptual) thing/perform one specific task.
- Functions should be as independent as possible, but this should not prevent hierarchical dependencies.
- Functions should not be overly long.
- Unless their primary function demands otherwise, most functions should return values instead of sending output to the console.

Declaring functions

Just as there is no keyword in C++ for declaring variables, there is no “function” keyword for declaring functions.

A function declaration is very similar to a variable declaration, but is distinguished by the parentheses of a (possibly empty) **formal parameter list**. The parentheses ‘(’ and ‘)’ are necessary even if the function has an empty parameter list.

A function declaration that occurs prior to any calls to that function is called a **forward declaration**.

In C++, functions must be declared before they can be called. Recall that the compiler reads source files linearly and tries to understand each ‘token’ as it goes along. (Class members are an exception.)

Syntax for declaring functions

Function syntax: declarations (prototypes)

```
‹returntype› ‹name› ();
‹returntype› ‹name› (‹datatype1›);
‹returntype› ‹name› (‹datatype1› ‹identifier1›);
‹returntype› ‹name› (‹datatype1›, ‹datatype2›);
‹returntype› ‹name› (‹datatype1› ‹identifier1›,
    ‹datatype2› ‹identifier2›);
// etc, with more parameters
```


Syntax explained

- ‹**name**› is a name used to identify this function, and should be reasonably descriptive as to the function’s purpose or behaviour.
- ‹**returntype**› is the type of value returned from the routine.
- ‹**datatype1**› specifies the type of the **formal parameter** ‹**identifier1**›, etc.
- Formal parameter identifiers need not be present in a declaration, and if there are multiple declarations, they need not match.
  However, using descriptive parameter identifiers increases the readability of the declarations.

A function declaration that is not also a definition does not have a function body, and is often called a **prototype** for that function.

Defining functions

A function definition is a function declaration that also has a **function body** (note that a definition also provides a declaration). The function body is an **implementation** of the function.

Function syntax: definitions

```
1 ‹returntype› ‹name› (‹datatype1› ‹identifier1›)
2 {
3     ‹statements›
4 }
```

- The function body is the portion between the curly braces.
- The curly braces **cannot** be omitted from function definitions.
- The curly brace ending the function body is **not** terminated with a semi-colon.

Function signatures

The **signature** of a function is its name, and the sequence of data types of its formal parameters.

> The names of the formal parameters do not matter.
>
> However, their types and their orderings do.
>
> Note: Parameter declarations that differ only in the presence or absence of `const` (and/or `volatile`) at the outer level are equivalent.

The signature is the information about a function that participates in **overload resolution**. More on this in another lecture. The compiler checks that there is an appropriate signature to use for each function call (e.g., when calling a function, that the right number of arguments are provided).

Function signatures do **not** include return type (except for template specialisations, discussed next term), or linkage type.

The `main` function

We have already seen some examples of functions.

Every program must contain a function '**main**'.

However, note that this function is unusual in many ways:

- same function can be defined with different prescribed signatures (but cannot be overloaded).
- technically need not have an explicit `return` statement (but still best to always have one).
- runs automatically when the program runs.
- the semantics of the **main** function is different for each program.

It is advised to make the **main** function just a wrapper for invoking functions with more descriptive names (with appropriate exception handling setup too), even if those new functions are only called once.

Example: `main` declaration

Function declaration:

```
int main();
```

Prototype: `int main();`
Return type: `int`
Identifier: **main**
Parameters: (empty)
Signature: **main**()
Header file: None
Implementation: Somewhere in your project, perhaps `main.cpp` or similar.

Example: `main` definition

Function definition:

```
int main()
{
    std::cout << "Hello world!" << std::endl;
    return EXIT_SUCCESS;
}
```

As per the declaration, but now have provided a **function body** (implementation) too.

Calling functions

The `main` function is the only function that automatically runs when it is defined.

All other functions must be **called** (**invoked**):

- in an expression (or expression statement), for functions that return values.
- in a statement, for functions with a `void` return type.

The syntax for calling a function whose signature is

```
‹name› (‹datatype1›)
```

is

```
‹name›(‹expression1›)
```

The expressions in the function call are **arguments** (or **actual parameters**), corresponding to the parameters in the function signature. Note that the parentheses are again mandatory even if the argument list is empty.

Function calls and program flow

- Function calls cause detours in the program flow.
- Recall: the program flow is the order in which statements are executed.
- A **call stack** is maintained to keep track of which functions are currently executing, what variables they are using, and where control flow is to return to when their executions complete.
- Functions only know about their own parameters and variables, not about those in any other function.
- This is better illustrated by example. . .

Functions: example

Example: A function

```
double compute_area_of_circle (double radius)
{
    const double PI = 3.14159;
    return PI * radius * radius;
}
```

Properties of the example function

Example: function

Prototype: `double compute_area_of_circle (double);`

Return type: `double`

Identifier: `compute_area_of_circle`

Parameters: `double radius`

Signature: `compute_area_of_circle(double)`

Header file: None; not used outside the translation unit in which it is defined.

Implementation: `circle_calculations.cpp` say

Functions: example, invoking

Example: Calling the function

The function on the previous slide can be called with something like:

```
6  int main()
7  {
8     double myRadius = 4.6;
9     double myArea = compute_area_of_circle(myRadius);
10    std::cout << "The area of a circle of radius " <<
         myRadius << " units is " << myArea << " square
         units." << std::endl;
11    return EXIT_SUCCESS;
12 }
```


Parameters vs. arguments

Parameters are the **variables** which are part of a function's interface.

Arguments are **expressions** used when calling a function.

In the previous example,

- The function `compute_area_of_circle` has a **parameter** `double radius`.
- The function was invoked (from the `main` function) with `myRadius` (`4.6`) as an **argument**.
- Due to scoping rules, the `compute_area_of_circle` function knows nothing about any identifier called `myRadius` (or `myArea`). Likewise, the `main` function knows nothing about any identifier called `radius` (or `PI`).

Functions can call functions. . .

Example

```
1  double square (double x)
2  {
3     return x * x;
4  }
5
6  double compute_area_of_circle (double radius)
7  {
8     const double PI = 3.14159;
9     return PI * square(radius);
10 }
```

The `return` keyword

The **`return`** keyword returns to the calling routine!

(That is, any remaining code between the **`return`** statement and the end of the function block is not executed — similar to **`break`** for iterative structures.)

Can use **`return`** multiple times in one function.

e.g. in different branches.

At most one **`return`** statement is ever actually executed per function call.

Returning values

For functions which return values (i.e. do not use the **`void`** return type):

- The **`return`** keyword is also used to specify the value returned to the calling function.
- It is not necessary to enclose the expression in parentheses.
- Each code branch must have a **`return`** keyword (though C++ may not enforce this, not having a **`return`** statement will result in undefined behaviour just as using a variable before assigning to it or initialising it).

The `void` return type

There will be times when you wish to define a function, but there is no natural value to return. For these cases, use the **`void`** return-type.

Such functions:

- need not have a **`return`** statement.
- can still exit the routine before the end by using **`return`** without any value.

Example: `void` return type

Example: `void`

```
1 void print_value (double val)
2 {
3     std::cout << "Value is: " << val << std::endl;
4 }
```

(N.B. here, **`val`** is arguably a candidate for a natural value to be returned.)

Example: parameterless function

The parameter list can be empty, eg

Example

```
1 void print_hello_world()
2 {
3     std::cout << "Hello world" << std::endl;
4 }
```

(For those of you that encounter C code, note that in C++, a function declared with an empty parameter list takes no arguments, whereas in C, an empty parameter list means that the number and type of the function arguments is not defined.)

Passing parameters to a function

There are two ways in which parameters can be passed to a function in C++, in general:

- By value: Pass a copy of the variable.
- By reference: Pass the variable itself (effectively), by means of a transparent memory address.

Which method is used is chosen by the function's author(s), in the formal parameter specifications. (The next slide describes how!)

Inside the function, changing the value of a formal parameter passed...

- ...by value: changes only the copy; doesn't change the original argument outside the function.
- ...by reference: **does** change the value of the original argument in the caller.

Passing by value vs. by reference

Passing by value is the default in C++ (including for objects — gets expensive as copies have to be made). Prefer to pass objects by const reference (or in certain situations, through pointers) instead.

In the `compute_area_of_circle` example, the variable `radius` (during this call) is a **copy** of the variable `myRadius` in the calling routine. `radius` also has a scope local to the `compute_area_of_circle` routine.

- This is called **passing by value**.
- Any changes made to `radius` would be lost at the end of the routine.

This is often the desired behaviour, but if the changes to `radius` need to be (automatically) passed back to the calling routine, we change the function to use **passing by reference**.

References

Can declare a variable as a reference: creates a synonym.

A variable is specified as a reference by using an ampersand (`&`) in the declaration.

Example

```
1 double usd_rate;
2 double &ref_to_usd_rate = usd_rate;
```

`ref_to_usd_rate` refers to the same memory location as `usd_rate`.

Reference types have to be initialised when they are defined, and what they point to cannot later be changed.

References (2)

Example of use

(Following the previous two lines.)

```
3 double x = 3.2;
4 ref_to_usd_rate = 6.0;
5 //usd_rate is now also 6.0
6 ref_to_usd_rate = x;
7 //usd_rate is now also 3.2
```

References are useful as a way of handling parameters in functions.

References

Note that once a reference has been declared, it behaves just like the underlying variable.

```
1 double x1=7.8, x2=9;
2 double& y = x1; //= here sets a reference
3 y = x2; //= here sets the underlying value
```


Lifetimes

Every automatic variable (local variable or temporary) has a defined lifetime. These lifetimes are often nested but basically never intersect. If *a* is created but not destroyed before *b* is created then *b* will be destroyed before *a*.

```
1  int add1(int x){
2      return x+1;
3  }
4  void foo(){
5      int x = 7, y = 9;
6      const int& z = 5 + (x+add1(y)); //the temporaries with value 10
       and 17 are created and destroyed within this line
7      //The reference causes the temporary with value 22 to last beyond its line.
8      x=99;
9      //At this point z,y, and x go out of scope. Their lifetimes end in that order.
10 }
```

References nicely fit in to this. If a reference is in scope, its underlying should be alive.

Passing by reference

This is where references are useful.

We use the **reference operator**, `&`, on the right of the datatype in the parameter list to specify those parameters that we wish to be passed by reference.

- For such parameters, the value of the corresponding argument is not passed to the function, only a reference to that argument.
- If passing a literal into a reference parameter, a temporary variable will be created.
- Can also specify parameters to be const references with `const &`.
  Then, if we attempt to modify the value, the compiler will stop us. This gives us a speed up compared to passing by value, but also protects us from modifying the value.
- Can pass `objects` (such as vectors) this way: much cheaper than by value.

Example: passing by reference

Example: passing by reference

```
1 void triple_value (double& val)
2 {
3    val *= 3.0;
4 }
```

This function is called in an identical way to those with by-value parameters:

```
5 double myVal = 5.0;
6 triple_value(myVal);
```

- **triple_value** does not return a value.
- The value of **myVal** after calling **triple_value** is now 15.0. Without the ampersand (&) on line 1, it would still be 5.0.
- **myVal** and **val** are (during this call) just two names for the same memory location.

References to temporaries

```
1 void triple_value (double& val){
2    val *= 3.0;
3 }
```

Logically, a non-const reference parameter, like **val** here, is an output. The language prevents a calculated value being passed here, as changes would be lost.

```
1 triple_value(3.0); //Not allowed
2 triple_value(3.0 * 4); //Not allowed
3 int& i = 47; //Not allowed
4 const int& i = 47; //OK
```

This rule is sometimes called "You cannot pass a temporary to a non-const reference". Rvalue references are a variation of references for which this rule does not apply, they are declared with two ampersands.

```
1 int&& i = 47; //OK
```


Return values from functions

In C++, a function can also return a reference (or a pointer).

- Do not return references (or pointers) to temporary/local variables.

  The memory used for such variables will be freed when the function returns, and it is invalid for the caller to attempt to access it.
- Only ever return a reference to something which outlives the function, e.g. a reference parameter, or part of a reference parameter.

Order of evaluation is unspecified

This is a valid program which will print 3, 4 and 5 in some order.

```
1 int p(int a){
2 cout<<a<<"\n";
3 return a;
4 }
5 void q(int a, int b){}
6 int main(){
7    q(p(3),p(4)+p(5));
8 }
```

But the order is "unspecified" - C++ does not say which order arguments to most operators and functions are evaluated (but they are all evaluated before the function). Even 3 may be printed between 4 and 5.

Extra

The following topics have not been covered:

- How C++ performs overload resolution
- Default values
- Recursive functions

Those people who are comfortable with functions already may wish to look these up on their own.

Program design and project layout

Designing programs

Before you start writing code, think about:

1. how to divide the program up into (objects and) functions, and
2. what data you need to pass around to those functions.

Think also about generalisations, but be aware that you can't expect to design a framework for solving the most general problem, as it is not easy to know what such a problem would look like, and even harder to design a suitable framework for it!

Modularisation

Write the program structure first.

Create functions to do self-contained things that you can fill in later.

- Can change the way the function works without changing the calling code.
- Can change the functionality without changing all the code.
- Can understand, improve and test functions independently.
- Can leave some parts of the implementation to do later, without having a gap in the middle of a core function. Just hive it off to another function!
- Other programmers can work on some of the functions.

Splitting source code into files

Just as it is a good idea to split a program into functions to enable separation and re-use of code **within** the program, it can also be helpful to split source code into files to enable separation and re-use of code **between** programs.

Even if there is no need to share the code between programs, a single source file can become too large (and slow) to manage, edit and compile effectively.

Splitting into translation units

Better not just to split into different **files**, but different **translation units**.

Generally, one `.cpp` file should correspond to a single translation unit.

Splitting source code into different translation units has several other advantages:

- **Compile decoupling**: You don't have to recompile ALL the code every time you make a change, only those translation units which (the build system thinks) have changed (and then relink).

A library is a collection of useful functions (and classes). Libraries are usually provided as collections of translation units, sometimes in object code form already.

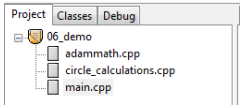
Compilation

Recall:

- Compiler compiles (separately) all translation units in the project into object code.
- Linker links all used object code (including that of any pre-compiled translation units/libraries) into one executable.

In order for the compiler and linker to know about all the translation units needed to produce a single executable when there is more than one, it is important to use a project in Dev-C++.

The source code for all the translation units (the `.cpp` files) should be part of the project.



Using code from other files

Recall that C++ needs to see forward declarations of other functions you are attempting to call before you call them.

When the implementations of those functions are in different translation units, need a way of providing those declarations without too much effort.

Header and implementation files

Every translation unit (except perhaps the one containing the `main` function) is usually split into two files:

1. a `.cpp` file, containing the implementation.
2. a `.hpp` file, containing the declarations of 'public' entities implemented by that translation unit ('public' meaning those entities that need to be accessible to other translation units). Such a file is called a **header file**.

Header files in C++ are usually saved with `.hpp` extension, and header files for the C language are saved with a `.h` extension, though not everyone observes this convention (many people use `.h` for C++ headers too).

Note that C++ system header files do not have an extension, but user header files should have one.

C and C++ header files

In C++, you should use:

```
#include <c[header]>
```

and fully qualify the symbol names you use with `std::`.

The standard does not require implementations to provide the symbols in the global namespace, only the `std` namespace.

In C, one uses:

```
#include <[header]>
```

and accesses the entities from the global namespace.

The versions without the 'c' prefix are for backwards compatibility **only**.

Using header files

- Header files and implementation files usually come in pairs and share the same base name, eg `circle_calculations.cpp` and `circle_calculations.hpp`.
- The `.cpp` file contains the implementation (the definitions) of the declarations in the header file.
- Any file (`.cpp` or `.hpp` etc) needing the prototypes from another translation unit should `#include` the corresponding header file.
- The `.cpp` file will often need the prototypes for its own functions etc, hence it may itself `#include` its corresponding header file.

Header file conventions

Header files are a convention only, and there are no restrictions on their content.

- They may contain more than just prototypes.
- Commonly also contain macro definitions.
- Are cases when they also include definitions. Discussed next term.
- Header files are usually **not** translation units in their own right.

Often, header files contain definitions as well as declarations.

In such cases, including the same header files multiple times in a translation unit would be an error.
Normally use **header guards** to allow a header file to be included more than once in a translation unit without causing errors (and also speeds up processing in that case).

Header guards

```
#ifndef CIRCLE_CALCULATIONS_HPP
#define CIRCLE_CALCULATIONS_HPP
// main header file code
#endif // CIRCLE_CALCULATIONS_HPP
```

Warning! Uses preprocessor macros (can be dangerous, but not considered evil for this purpose).

Need to use a unique identifier **`CIRCLE_CALCULATIONS_HPP`** for every header file AND ensure that the identifier is not used anywhere else in the project.

Preprocessor include directives

Recall that `#include` is a preprocessor directive.

Examples

```
#include <cstdlib>
#include "circle_calculations.hpp"
```

- The difference between `#include` with `<>` and with `""` is implementation-defined, but generally `<>` searches only the system paths while `""` searches the directory containing the current file first and then the system paths.
- In general, use `<>` for system includes, and `""` for includes local to your program.

Navigating to header files

Although header files need not be part of the project, still need to view and edit them.

If they are not part of the project tree

how do you do it from within the IDE?

In Dev-C++, from the context menu with the `.cpp` file selected, choose “Swap header/source”.

Lab objectives

- Know how to define a function and specify parameters and return values.
- Appreciate how functions make life easier.

The exercises build on earlier ones.
Even if you have not written the non-function versions, please jump straight into writing the function versions now.