C++ Tutorial

College of William & Mary
Last updated: June 7, 2013

About this tutorial

This set of exercises was written for the Research Experiences for Undergraduates (REU) program at the College of William & Mary. The exercises that can be worked through at any pace that you are comfortable with. You will learn by trying to find the solutions to these exercises (in this document, but more importantly by using other resources identified below). Exercises to be completed as part of this tutorial are highlighted in yellow.

Along with this set of exercises you may want to use the C++ Language Tutorial at http://www.cplusplus.com/doc/tutorial/. The relevant sections will be indicated for each problem.

Download and/or open the C++ Language Tutorial for reference throughout this tutorial.

Servers, editors, and other software

In this tutorial we will focus on C++ development in a linux environment with the g++ compiler. However, there are many other compilers and programming environments.

Integrated development environments

In an integrated development environment (IDE) the compiler, editor and debugger are integrated in the same software program. This typically makes for a more productive environment when you can spend the time in setting it up to work according to your needs.

- **Eclipse** (with C/C++ Development Tooling), http://www.eclipse.org/cdt/
  This is an integrated development environment originally developed for Java programming but excellent support for C/C++ is available as well. It has syntax highlighting, auto-completion and even auto-generation of code, and it includes a debugger.
- **Kdevelop**, http://www.kdevelop.org/
  This integrated development environment focuses on C/C++ development, in particular for open source projects (e.g. KDE). It includes a similar set of features as Eclipse above.

- **Visual C++ Express** (part of Visual Studio Express), http://www.microsoft.com/visualstudio/eng/team-foundation-service
  This is an integrated development environment for Microsoft Windows.

Editors in linux

Because you will often find yourself editing files while logged in on remote systems, text-based editors maintain their role in editing C++ code. Graphical editors are slower over network connections, unless there is some sort of caching mechanism for remote files.

- **nano**: a bare-bones text-based editors with basic syntax highlighting (after some effort). Exit by typing ‘Ctrl-x’.
- **vi**: a text-based editor with a fairly steep learning curve. Exit by typing ‘:q’.
- **emacs**: a graphical or text-based editors with a fairly steep learning curve. Exit by typing ‘Ctrl-x,Ctrl-c’.
- **nedit**: a basic graphical editor with syntax highlighting (after some effort). Nedit runs smoothly over ssh connections.
- **kate**: a graphical editor with syntax highlighting. Kate supports opening files over ssh connections using the ‘fish://server/path/to/file’ syntax.

Logging into the tutorial server

Use your favorite ssh client to log into **sporades.physics.wm.edu** as user **RootTutorial**. The password will be given to you during the tutorial.

```
ssh -X RootTutorial@sporades.physics.wm.edu
```

Copy the set of tutorial files to a local directory. During the rest of this tutorial you will work only in this local directory. In the following commands `<username>` should be your W&M username.

```
mkdir ~/users/<username>
cp -r ~/cplusplus_tutorial_files ~/users/<username>
cd ~/users/<username>/cplusplus_tutorial_files
```

Familiarize yourself with your text editor of choice. If you aren’t familiar with any particular editor you may want to use `nano`. Just enter, for example, `nano helloworld.cpp` to open the file `helloworld.cpp`.

Structure of a program

Source and header files

There are two important types of files in C++ programs:

- **Source** or **implementation** files, typically with the extension `.cpp`, contain the actual program code.
- **Header** or **interface** files, typically with the extension `.h` or `.hpp` contain the interface to the code in the source file. Only the header files need to be distributed for others to be able to write programs that use your code.
    - Standard header files, such as `iostream` for input and output streams, are distributed with your compiler collection and typically have no extension. They can be used with `#include <iostream>`.
    - Header files that you wrote can be included with `#include "helloworld.h"`.

Open the files `helloworld.cpp` and `helloworld.h`. Make sure you understand the function of each line by comparing with the section Structure of a program in the C++ Language Tutorial.

Compilation

Contrary to other programming languages that you may be familiar with, C++ typically requires that you compile the programming code to turn it into an executable file. We will use the **GNU compiler collection** (http://gcc.gnu.org/) which includes the **g++** compiler.

To compile the source code in `helloworld.cpp` and `helloworld.h`, execute the following command

```
g++ -Wall -o helloworld helloworld.cpp
```

This will instruct the compiler, `g++`, to generate an output file (`-o`) with the name `helloworld` based on the source files that follow (in this case just `helloworld.cpp`, which refers to `helloworld.h` so we do not need to specify it again). The option `-Wall` tells the compiler to warn (`-W`) about every (`all`) possible issue with your code (strive for code without warnings, and only warnings you absolutely, completely, positively understand).

You can check that the 'execute bits' are set on the file `helloworld` by executing `ls -al`, which will list all files in your current directory.

Finally, execute the program you just compiled.

```
./helloworld
```

You need to specify the `./` to make explicitly clear that you want the executable `helloworld` in the current directory (`.`), and not elsewhere on the system.

Exercises

- Introduce the following errors into your program and attempt to compile. Take note of the errors or warnings that appear. You will encounter them many more times... **No need to be afraid of compiler errors!**

   - Change the name of the file in the `#include` statement to a non-existing file.
   - Change the name of the `main` function. The function that is called when you execute a program must have the name `main`.
   - Remove the semicolon at the end of the “Hello, world!” line.
   - Remove the 0 in the `return` statement.
   - Remove the line `using namespace std;` in the header file. This error can be resolved by explicitly specifying `std::cout` and `std::endl` in the source file. The objects `cout` and `endl` are part of the namespace `std`.
- Modify the helloworld program to print “Goodbye, world!”. Do not remove the line “Hello, world!” but comment it so you can enable it again later.
- Remove the need for the header file by moving all directives into the source file. Verify that the compilation still succeeds.

Variable and data types

Data types

The most common data types are: `bool`, `char`, `int`, `float`, `double`, and `std::string`. The last one, `std::string`, is not a basic data type (it is in the `std` namespace and has its own header file `string`), but it is fundamental nonetheless and preferred over the C approach of using arrays of `char`s.

Later in this tutorial we will discuss classes, one of the core concepts in C++. Classes are in many ways an extension of data types.

Declaration and initialization

It has been said that computers are the dumbest things on the planet. In C++ this certainly applies to the data types of variables: you have to be very explicit about which data type a variable will contain. Other languages may do this for you (and, for example, automatically assume that the `i` in `i = 1` can only assume integer values), but in C++ it is completely up to you. To do this you need a **declaration** of the data type.

Only after the variable has been declared can you use it to hold a value. You will need to **initialize** the variable with the first value it will hold.

```
int a; // declaration of variable a to be of data type int
a = 1; // initialization of variable a to the value 1
```

These statements can be combined, and another syntax can be used:

```
int a = 1; // combined declaration/initialization of variable a
int b(a); // declaration/initialization of variable b
```

The syntax with the parentheses will come back when we talk about classes as a more efficient way to initialize new variables. In the equivalent of `a = 1` for classes a temporary object is created to hold 1 and that temporary object is assigned to `a`. In the equivalent syntax `a(1)` that temporary object does not need to be created.

If you assign a variable of a different data type to a variable, unexpected things can happen: a floating point number can get truncated, or (even worse) a string of letter can be turned into a number that is complete nonsense.

In the source file `datatypes.cpp` assign the number 5.1 to the integer variable `a` and see what happens.

Using the source file `datatypes.cpp` and the section Variables. Data Types in the C++ Language Tutorial determine the ratio (a floating point number!) of the two integer numbers `a = 355` and `b = 113`.

Scope

Variable can only be used in their 'scope': the smallest block of braces (`{ /*...*/ }`) that surrounds them. Outside the block the variable is simply not known (unless you declare another variable with the same name) and the compiler will complain. It is good practice to declare variables only with the smallest scope needed. This means that you should only declare variables when you need them and where you need them.

Add a block of braces that only surrounds the declaration line for `b` in the source file `datatypes.cpp`. Verify that compilation fails due to these scope rules.

Type casting

To explicitly change from one data type to another data type you need to 'type cast'. For example, you can 'cast' a floating point number into an integer (by truncation) using the following two styles:

- C style casting: `int a = (int) b; // discouraged`
- C++ style casting: `int a = dynamic_cast<int>(b);`

The C++ style is preferred because it works with classes as well.

If you want to type cast an int into a number with a decimal (float or double) do something like:

- C++ style casting: double a = static_cast<int>(b);

For changing an integer into a decimal number use double rather than float, because float may not have enough memory to hold the the integer.

Arrays and vectors

Often we need to store not just a single number but an entire list, or even a two-dimensional matrix. To do this we use arrays, declared as follows:

```
int a[3] = {0, 1, 2}; // array of 3 integers with initialization
int b[] = {0, 2}; // without explicit size (as it's redundant)
int M[2][2]; // 2 x 2 matrix without initialization
```

To retrieve the values by index we use (counting from zero for the first element!)

```
cout << a[0] << endl; // first element in a
cout << M[1][1] << endl; // second diagonal element in M
```

The initialization should only be used for small, 1-dimensional arrays as it becomes unwieldy very quickly.

Arrays have several disadvantages: you cannot determine the length of an array from the array itself, it is difficult to add elements or to resize them, etc. Therefore it is more useful for all but the simplest of cases to use the `vector` functionality in C++. In this context a `vector` can contain anything, not just integers but also classes (see later).

```
#include <vector>

vector<int> a(3); // vector of integers with size 3
// Unfortunately, the initializer lists {0,1,2} does not work
// with these vectors.

cout << a.size() << endl; // print size of a

vector<vector<int> > M; // vector of vectors of integers

M.resize(2); // resize M to have two rows
M[0].resize(2); // two elements in first row
M[1].resize(3); // three elements in second row
cout << M[0].size() << endl; // print size of first row of M
cout << M[1].size() << endl; // print size of second row of M
```

Modify the file `arrays.cpp` to use `vector<int>` arrays.

Control structure (if/else, for, switch)

A programming language wouldn't be much without control structures (conditional statements and loops).

`if/else`

Conditional statements in C++ are written as follows:

```
if (a == b) {
  /* do something */
} else {
  /* do something else */
}
```

for

For-loops in C++ are written as follows:

```
for (int i = 0; i < 10; i++) {
  /* do something */
}
```

There are three statements in the parentheses. The first statement (`int i = 0`) is executed before beginning the loop. The second statement (`i < 10`) is the condition that needs to be true to continue the next iteration in the loop. The third statement (`i++`) is executed at the end of each iteration; in this case it just increments `i` by 1. This loops starts with `i` = 0, and loops for 10 times until in the last iteration `i` = 9.

When looping over `vector`s you will typically encounter structures like this

```
for (size_t i = 0; i < v.size(); i++) {
  /* do something with v[i] */
}
```

The reason for using `size_t` instead of `int` is to ensure that `i` has the same data type as what `v.size()` returns.

Write a `for`-loop that loops over all entries in `billy` in the file `arrays.cpp` after your modifications to use `std::vector` above.

switch

A convenient shorthand for many `if/else` statements is the `case/switch` statement:

```
switch (a) {
  case 1:
    /* do something if a = 1 */
    break;
  case 2:
    /* do something if a = 2 */
    break;
  default:
    /* do something in all other cases */
}
```

In this `switch` statement, if `a = 1` then it will do the statement after `case 1`, if `a = 2` then it will do the statement after `case 2`. This can be extrapolated to other data types, for example, with a data typed defined as Color the variable, eColor can be used in a switch statement like:

```
switch (eColor)
    {
        case COLOR_BLACK:
            cout << "Black";
            break;
        case COLOR_WHITE:
            cout << "White";
            break;
        case COLOR_RED:
            cout << "Red";
            break;
        case COLOR_GREEN:
            cout << "Green";
            break;
        case COLOR_BLUE:
            cout << "Blue";
            break;
        default:
            cout << "Unknown";
            break;
    }
```

Switch statements can only have the variable equal to one thing. If you want it equal to something in a range or something "and-ed" or "or-ed" it is best to use if or else statements.

Rewrite the code in `switch.cpp` to use `case/switch` instead of `if/else` statements.

Functions (Functions I and II in C++ Language Tutorial)

In C++ functions are defined by the following format

```
type name (parameter1, parameter2,...) {
  /* statements */
}
```

The important parts are the return data type `type`, and the parameters with type and name (for example, `parameter1` could be `int a`). When a function does not return anything (and for example only prints something to the screen) we can use the return typ `void`.

Signatures

Just as variables need a declaration, function can (but don't have to) have a **signature** before they are defined. The signature for a general function (as above) is

```
    type name (parameter1, parameter2,...);
```

This signature will typically be placed in the header file, as it provides the **interface** to using the functions that are defined in the source file.

Create a set of files `subtraction.cpp`, `subtraction.h`, `addition.cpp`, `addition.h` and `test_addition_subtraction.cpp` where the main function resides in `test_addition_subtraction.cpp` while all other files contain just a function implementation (`.cpp`) or interface (`.h`). Compile this by specifying all `.cpp` files on the command line with a single executable file as output (`-o`).

Call by value

When a function such as `int addition (int a, int b)` in `addition.cpp` is called, the two arguments are copied and the function operates on the copy only. The original values of the arguments are not changed.

Verify that having the function as call by value in the file `addition.cpp` does indeed **not** modify the value of `x` and `y`.

Call by reference

It is also possible to call the function as `int addition (int& a, int& b)`. The additional `&` characters indicate that the values for `a` and `b` are **not** copies but are the original values. The variables `a` and `b` are now **references** to the original variables, and any changes to `a` and `b` will affect the original values.

Verify that changing the function to call by reference in the file `addition.cpp` does indeed modify the value of `x` and `y`.

Pointers

A special 'derived' data type in C++ is the pointer. It is a variable that contains the **memory address of another variable**. For details, please read the section Pointers in the C++ Language Tutorial.

The two important operations for pointers are:

- referencing: pointer = **&** variable
- dereferencing: value = ***** pointer

Here is an example of their use:

```
int a = 1; // integer variable, initialized to 1
int* p = &a; // pointer p with address of variable a
cout << *p << endl; // value at address in pointer p
```

Modify the function `addition` in the file `addition.cpp` to take pointers to integer numbers instead of integer numbers. This is equivalent to call by reference, so changing the value of the pointers in the function should change the original values as well.

Using pointers allows you to explicitly allocate memory upon declaration and initialization.

```
int a; // regular variable declaration
int b[3]; // regular array declaration
int* pa = new int; // allocate memory for one int pointer
int* pb = new int[3]; // allocate memory for an int array
```

Again, this will prove very useful when using classes.

Classes

Classes are extensions of the basic data types. Here is an example of a class that defines a rectangle with two sides of length `x` and `y`. This is just the abstract object that **represents** a rectangle, so it will not pop up on the screen as a graphical rectangle (until you program in the necessary functionality to do that).

```
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void);
};
```

There are `private` and `public` blocks of class data members (or fields) and functions (or methods). The `private` members and functions can only be accessed by the objects of the class itself. The `public` members and functions can be accessed by everyone.

To create a variable of the type `CRectangle` you can use the same syntax as for the declaration of basic data types:

```
CRectangle rect; // the variable rect of type CRectangle
```

On the object `rect` you can now call all public functions and access public data members:

```
rect.set_values (3,4); // set the values on object rect
myarea = rect.area(); // get the area
```

Notice that we have not yet specified the implementation for the functions `void set_values(int,int)` and `int area (void)`. For example

```
void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}
int CRectangle::area (void) {
  return (x*y);
}
```

The notation `::` indicates that the functions `set_values` and `area` are **inside** the class `CRectangle`.

Constructor/destructor

C++ knows how to declare and initialize basic data types, but it doesn't know how to create an object of the type `CRectangle` so we have to tell it. Similarly, when an object goes 'out of scope' the memory occupied by a variable will be released. We have to tell C++ how to do that for our own classes. This is done through two special functions: the constructor and destructor function (without return type):

```
CRectangle::CRectangle();
CRectangle::~CRectangle();
```

What's "this"?

There is one special pointer, the `this` pointer that points to the object itself. For example,

```
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void);

    void print_values() {
      cout << this->x << " x " << this->y << endl;
    }
};
```

The `this` pointer is necessary when using inheritance (see below) and when you want to call functions defined in the class that you are inheriting from.

Inheritance

The real power of C++ (yes, only on page 11) lies in the inheritance of classes. You can **inherit** the functionality of a class into a new class. For example:

```
class CBox: public CRectangle {
  // inherits everything from CRectangle to represent the base
    int z; // height of the box
  public:
    void set_height(int);
    int volume(void); // volume of the box
};
```

Implement the class CBox that inherits from CRectangle (in separate files) and write a simple test program. You will need to implement constructors and destructors, and the functions specific to `CBox`. You will need to use the `this` pointer to access the `area()` of the rectangle.

Templates

In the previous sections we defined a rectangle and box class that has sides with integer lengths. This may have been sufficient in ancient times, but by now we have discovered rectangles with sides with floating point lengths. To satisfy both the ancient Greeks and the modern mathematicians we want to make the data type for the lengths of the sides variable somehow. This is accomplished by **templates**.

```
template<T>
class CRectangle {
    T x, y;
  public:
    void set_values (T,T);
    T area (void);
};

CRectangle<int> rect_with_int_lengths;
CRectangle<double> rect_with_double_lengths;
```

Everywhere where `T` appears the compiler will replace it with the type you specify on declaration of the actual variable. We have actually used this earlier when we discussed `vector<int>`.

Rewrite the `CRectangle` class to use templates and test whether the area is now indeed a floating point (or double precision) number.

Don’t forget proper coding etiquette

- comment your code, It will be useful to
    - you when you look back at it a few weeks, months, or years from now
    - someone else if they pick up your code or work on it
    - also letting people know the purpose of the program you are working on and what functions in your program do is really appreciated