

CSCI-1200 Data Structures — Fall 2019

Lecture 0 — Introduction to C++

Instructor

Professor Barb Cutler, cutler@cs.rpi.edu
Lally 302, 518-276-3274

Instructional Support Coordinator

Shianne Hulbert, hulbes@rpi.edu
Amos Eaton (AE) 109, 518-276-6911

To ensure a timely response, please use: ds_instructors@cs.rpi.edu

Today

- Brief Discussion of Website & Syllabus
<http://www.cs.rpi.edu/academics/courses/fall19/csci1200/>
- Crash Course in C++ Syntax
http://www.cs.rpi.edu/academics/courses/fall19/csci1200/crash_course_cpp_syntax.php
- C++ Development Environment Installation
http://www.cs.rpi.edu/academics/courses/fall19/csci1200/development_environment.php

0.1 Transitioning from Python to C++ (from CSCI-1100 Computer Science 1)

- Python is a great language to learn the power and flexibility of programming and computational problem solving. This semester we will work in C++ and study lower level programming concepts, focusing on details including efficiency and memory usage.
- Outside of this class, when working on large programming projects, you will find it is not uncommon to use a mix of programming languages and libraries. The individual advantages of Python and C++ (and Java, and Perl, and C, and UNIX bash scripts, and ...) can be combined into an elegant (or terrifyingly complex) masterpiece.

0.2 Compiled Languages vs. Interpreted Languages

- C/C++ is a *compiled language*, which means your code is processed (compiled & linked) to produce a low-level machine language executable that can be run on your specific hardware. You must re-compile & re-link after you edit any of the files – although a smart development environment or **Makefile** will figure out what portions need to be recompiled and save some time (especially on large programming projects with many lines of code and many files). Also, if you move your code to a different computer you will usually need to recompile. Generally the extra work of compilation produces an efficient and optimized executable that will run fast.
- In contrast, many newer languages including Python, Java, & Perl are *interpreted languages*, that favor incremental development where you can make changes to your code and immediately run all or some of your code without waiting for compilation. However, an interpreted program will almost always run slower than a compiled program.
- These days, the process of compilation is almost instantaneous for simple programs, and in this course we encourage you to follow the same incremental editing & frequent testing development strategy that is employed with interpreted languages.
- Finally, many interpreted languages have a Just-In-Time-Compiler (JIT) that can run an interpreted programming language and perform optimization on-the-fly resulting in program performance that rivals optimized compiled code. Thus, the differences between compiled and interpreted languages are somewhat blurry.
- You will practice the cycle of coding & compilation & testing during Lab 1 & Homework 1. You are encouraged to try out different development environments (code editor & compiler) and quickly settle on one that allows you to be most productive. Ask the your lab TAs & mentors about their favorite programming environments! The course website includes many helpful links as well.
- As you see in today's handout, C++ has more required punctuation than Python, and the syntax is more restrictive. The compiler will proofread your code in detail and complain about any mistakes you make. Even long-time C++ programmers make mistakes in syntax, and with practice you will become familiar with the compiler's error messages and how to correct your code.

0.3 A Sample C++ Program: Find the Roots of a Quadratic Polynomial

```
#include <iostream>    // library for reading & writing from the console/keyboard
#include <cmath>        // library with the square root function & absolute value
#include <cstdlib>      // library with the exit function

// Returns true if the candidate root is indeed a root of the polynomial  $a*x*x + b*x + c = 0$ 
bool check_root(int a, int b, int c, float root) {
    // plug the value into the formula
    float check = a * root * root + b * root + c;
    // see if the absolute value is zero (within a small tolerance)
    if (fabs(check) > 0.0001) {
        std::cerr << "ERROR:  " << root << " is not a root of this formula." << std::endl;
        return false;
    } else {
        return true;
    }
}

/* Use the quadratic formula to find the two real roots of polynomial.  Returns
true if the roots are real, returns false if the roots are imaginary.  If the roots
are real, they are returned through the reference parameters root_pos and root_neg. */
bool find_roots(int a, int b, int c, float &root_pos, float &root_neg) {
    // compute the quantity under the radical of the quadratic formula
    int radical = b*b - 4*a*c;
    // if the radical is negative, the roots are imaginary
    if (radical < 0) {
        std::cerr << "ERROR:  Imaginary roots" << std::endl;
        return false;
    }
    float sqrt_radical = sqrt(radical);
    // compute the two roots
    root_pos = (-b + sqrt_radical) / float(2*a);
    root_neg = (-b - sqrt_radical) / float(2*a);
    return true;
}

int main() {
    // We will loop until we are given a polynomial with real roots
    while (true) {
        std::cout << "Enter 3 integer coefficients to a quadratic function:  $a*x*x + b*x + c = 0$ " << std::endl;
        int my_a, my_b, my_c;
        std::cin >> my_a >> my_b >> my_c;
        // create a place to store the roots
        float root_1, root_2;
        bool success = find_roots(my_a, my_b, my_c, root_1, root_2);
        // If the polynomial has imaginary roots, skip the rest of this loop and start over
        if (!success) continue;
        std::cout << "The roots are: " << root_1 << " and " << root_2 << std::endl;
        // Check our work...
        if (check_root(my_a, my_b, my_c, root_1) && check_root(my_a, my_b, my_c, root_2)) {
            // Verified roots, break out of the while loop
            break;
        } else {
            std::cerr << "ERROR:  Unable to verify one or both roots." << std::endl;
            // if the program has an error, we choose to exit with a
            // non-zero error code
            exit(1);
        }
    }
    // by convention, main should return zero when the program finishes normally
    return 0;
}
```

0.4 Some Basic C++ Syntax

Crash Course in C++: Lesson #2

- Comments are indicated using `//` for single line comments and `/*` and `*/` for multi-line comments.
- `#include` asks the compiler for parts of the standard library and other code that we wish to use (e.g. the input/output stream function `std::cout`).
- `int main()` is a necessary component of all C++ programs; it returns a value (integer in this case) **and** it may have parameters.
- `{ }`: the curly braces indicate to C++ to treat *everything* between them as a unit.

0.5 The C++ Standard Library, a.k.a. “STL”

Crash Course in C++: Lesson #3

- The standard library contains types and functions that are important extensions to the core C++ language. We will use the standard library to such a great extent that it will feel like part of the C++ core language. `std` is a *namespace* that contains the standard library.
- I/O streams are the first component of the standard library that we see. `std::cout` (“console output”) and `std::endl` (“end line”) are defined in the standard library header file, `iostream`

0.6 A few notes on C++ vs. Java

- In Java, everything is an object and everything “inherits” from `java.lang.Object`. In C++, functions can exist outside of classes. In particular, the `main` function is never part of a class.
- Source code file organization in C++ does not need to be related to class organization as it does in Java. On the other hand, creating one C++ class (when we get to classes) per file is the *preferred* organization, with the *main* function in a separate file on its own or with a few helper functions.

0.7 Variables and Types

Crash Course in C++: Lesson #1

- A variable is an object with a name. A name is C++ identifier such as “a”, “root_1”, or “success”.
- An object is computer memory that has a type. A type (e.g., `int`, `float`, and `bool`) is a structure to memory and a set of operations.
- For example, a `float` is an object and each `float` variable is assigned to 4 bytes of memory, and this memory is formatted according IEEE floating point standards for what represents the exponent and mantissa. There are many operations defined on floats, including addition, subtraction, printing to the screen, etc.
- In C++ and Java the programmer must specify the data type when a new variable is declared. The C++ compiler enforces type checking (a.k.a. *static typing*). In contrast, the programmer does not specify the type of variables in Python and Perl. These languages are *dynamically-typed* — the interpreter will deduce the data type at runtime.

0.8 Expressions, Assignments, and Statements

Crash Course in C++: Lesson #2

Consider the *statement*: `root_pos = (-b + sqrt_radical) / float(2*a);`

- The calculation on the right hand side of the `=` is an expression. You should review the definition of C++ arithmetic expressions and operator precedence from any reference textbook. The rules are pretty much the same in C++ and Java and Python.
- The value of this expression is assigned to the memory location of the float variable `root_pos`. Note also that if all expression values are type `int` we need a *cast* from `int` to `float` to prevent the truncation of integer division.

0.9 Conditionals and IF statements

Crash Course in C++: Lesson #4

- The general form of an if-else statement is

```
if (conditional-expression)
    statement;
else
    statement;
```

- Each **statement** may be a single statement, such as the `cout` statement above, or multiple statements delimited by `{...}`.

0.10 Functions and Arguments

Crash Course in C++: Lesson #9

- Functions are used to:
 - Break code up into modules for ease of programming and testing, and for ease of reading by other people (never, ever, under-estimate the importance of this!).
 - Create code that is reusable at several places in one program and by several programs.
- Each function has a sequence of parameters and a return type. The function *prototype* below has a return type of `bool` and five parameters.

```
bool find_roots(int a, int b, int c, float &root_pos, float &root_neg);
```
- The order and types of the parameters in the calling function (the main function in this example) must match the order and types of the parameters in the function prototype.

0.11 Value Parameters and Reference Parameters

Crash Course in C++: Lesson #10

- What's with the `&` symbol on the 4th and 5th parameters in the `find_roots` function prototype?
- Note that when we call this function, we haven't yet stored anything in those two root variables.

```
float root_1, root_2;  
bool success = find_roots(my_a, my_b, my_c, root_1, root_2);
```
- The first three parameters to this function are *value parameters*.
 - These are essentially local variables (in the function) whose initial values are *copies* of the values of the corresponding argument in the function call.
 - Thus, the value of `my_a` from the main function is used to initialize `a` in function `find_roots`.
 - Changes to value parameters within the called function do NOT change the corresponding argument in the calling function.
- The final two parameters are *reference parameters*, as indicated by the `&`.
 - Reference parameters are just aliases for their corresponding arguments. No new objects are created.
 - As a result, changes to reference parameters are changes to the corresponding variables (arguments) in the calling function.
- In general, the “Rules of Thumb” for using value and reference parameters:
 - When a function (e.g., `check_root`) needs to provide just one result, make that result the return value of the function and pass other parameters by value.
 - When a function needs to provide more than one result (e.g., `find_roots`, these results should be returned using multiple reference parameters.
- We'll see more examples of the importance of value vs. reference parameters as the semester continues.

0.12 for & while Loops

Crash Course in C++: Lesson #5

- Here is the basic form of a `for` loop:

```
for (expr1; expr2; expr3)  
    statement;
```

 - `expr1` is the initial expression executed at the start before the loop iterations begin;
 - `expr2` is the test applied before the beginning of each loop iteration, the loop ends when this expression evaluates to `false` or 0;
 - `expr3` is evaluated at the very end of each iteration;
 - `statement` is the “loop body”
- Here is the basic form of a `while` loop:

```
while (expr)  
    statement;
```

`expr` is checked before entering the loop and after each iteration. If `expr` ever evaluates the false the loop is finished.