Lecture 29

Introduction to C++

We begin our introduction to C++.

C++ a quick history

- C was created in 1972. Back then, computers were slow, didn't have a lot of memory and were shared.

  In fact, it was typical to program at a "dumb terminal" which was basically a monitor and keyboard connected to a server through a wired connection. So slow, that original editors were line editors, so that they didn't have to draw whole document on screen.

- Because of this C was designed to be fast! Sacrifices were made with respect to security, bounds and type checking, etc.

- C++ is a new language based on C. Crucially, all C++ was supposed to be backward compatible with C. That is, any C program should compile and run as a C++ program.

- C++ was meant to be a replacement for C, however, since the designers were concerned that people wouldn't switch if their programs got slower, C++ made many of the same sacrifices as C.

  A lot of the differences between C++ and Java that we'll talk about for the rest of the semester are either direct or indirect result of these decisions.

- C++ has very similar syntax to Java. This is because Java borrowed much of the simple syntax of C to make it easy for programmers to switch.

  This will be both good and bad. It will make it easier to learn C++ if we already know Java. But it may be confusing to see the same instructions have slightly different behavior.

Why C++?

Why do we still study C++? It is still a very relevent language.

- Still very popular language.
- Very fast (though Java has been closing the gap for years).
- Allows us to interact directly with hardware (operating systems and device drivers).
- Offers more control (direct access to memory, devices).

Available Resources

See the handouts section of the class web page for links to C++ resources, documentation, tutorials, etc.

Our first C++ program

Examine the file named `first.cpp` from the code webpage for this lecture.

What are some of the similarities with Java:

- First, it has a `main` method. Like Java, execution begins in the `main`, and an executable only has one.
- primitive types. very similar. Java has 8 – `short, int, long, double, float, boolean, char,` and `byte`.

  C++ has many of the same primitives – `short, int, char`. In C++ they can be modified as "`signed`" or "`unsigned`." Also, many can be declared "long."
- Library types. C++ has extensive standard library with many of the same types as Java.
- Similar syntax. Curly braces. method construction (parameters, return types, name).
- Control constructs are basically the same – `for` loops, `while` loops, `if..if-else`, etc.
- Use of generics. C++ has had generics for a long time, Java only got them recently with Java 5.
- Calling methods of the object with the '.' operator.

Even in this simple program, there are some noticeable differences:

- no class! C++ doesn't require one (but does have them, of course). Java needs a class no matter what!
- main method – simpler method header. no arguments, return type `int`. There are other versions of the `main` function in C++ that we'll see later.
- `#include` statements. These may look similar to Java's import statement, but they're really not. They are the equivalent of cutting and pasting the mentioned file at the include location.

  When we say `#include <file>`, we are using a built in system file.

  When we say `#include ''file''` we are referring to a user class file, and we must give an appropriate path name.

  In fact, anything starting with "#" sign in C/C++ is a pre-processor directive. This means that these instructions are performed *prior to* compilation.
- `using std namespace`. This **is** the equivalent of Java's import statement.

  Remember (maybe?) in Java, that import makes our life easier. If we want to use Java's `ArrayList` class, then we can import `java.util.ArrayList`. Now we can declare new variables like this:

  ```
  ArrayList<String> list = new ArrayList<String>();
  ```

  Java imports "java.lang" by default, so we don't have to put it in our code. C++ has no such default. We always have to specify.

  In C++, the `vector` type is in the "std" namespace. By saying

  ```
  use std namespace
  ```

  This is like "import java.util.*;" So we can just create a new vector like this:

  ```
  vector<int> nums;
  ```

  Without the "using" command, we refer to types in the library as follows:

  `std::vector` or `std::cout`

  The `::` operator is known as the *scope resolution operator*.

  The sample file `first_no_using.cpp` gives a C++ file that is equivalent to `first.cpp`, but uses only the scope resolution operator to avoid using the `using namespace` command.

- Function prototypes. In Java, we could place our method definitions anywhere in the class file. C++ requires that we have seen the declaration of a function or variable before we use it, so it must appear before it in the file. There are times when this is difficult, or even impossible (mutual recursion). To handle these cases, C++ allows *function prototypes*. Function prototypes allow us to declare the name before we implement the code, so that we can refer to the function name before we implement it.

- Operator overloading. Notice the "`nums[i]`" notation. `nums` is of type `vector`, which is just a type like the Java `Vector` or `ArrayList`.

  The C++ documentation for `vector` shows the "operator[]" entry in the member function list. This is operator overloading. We have overloaded the [] operator, so that it acts like `at` member function. The `at` function just returns the value at a particular index in the array.