Introduction to C++

CS 2ME3/SE 2AA4

Steven Palmer

Department of Computing and Software
McMaster University

February 26, 2018

Outline

C++

- C++ is a language based on C – started as an Object Oriented extension to C (originally called "C with Classes")
- Backwards compatible with C – any C program will compile (or be made to compile) with C++
- Modern C++ standards have added many additional constructs to the language – both OO related and not
- As a result, C++ is a **very big** language – this tutorial will cover some basic syntax and concepts you will need for the assignments
- You will almost certainly need to do some reading/practice on your own to fill in the gaps – references to good resources are provided on the last slide

gcc (the GNU Compiler Collection)

- In this course we will use g++ (part of the GNU Compiler Collection – gcc) to compile C++ code
- Installation:
  - Windows: via MinGW
  - Mac:
    - by default `gcc/g++` is probably available, but this is actually an alias for `clang`
    - use homebrew or macports to get proper `gcc/g++`
  - Linux:
    - likely installed by default
    - if not check your package manager

File Organization

- C++ uses header files (extension .h) and source files (extension .cpp)
- Generally, for every source file you write, you will write an accompanying header file
- Header files contain declarations of variables, classes, and functions
- Source files contain the definitions of the things defined in the header
- More on this later

Base Types and Derived Types

- C++ includes the standard base types that you are familiar with: int, float, double, bool, char, etc.
- There is an additional base type called void which is used as the type for functions which do not return anything (there are additional uses of void, but those are beyond the scope of this course)
- Note that strings are not a base type in C++
- We can build additional derived types using various language constructs (we will focus on enum and class – discussed later)

Functions

- Function definitions in C++ use the following syntax:

```
type functionName(type p1, type p2, ...) {
  function body
}
```

- For example, a function that returns the smallest of 2 integers would be:

```
int min(int a, int b){
  if (a < b)
    return a;
  return b;
}
```

Importing Modules

- We can import modules in C++ by using the include directive (#include)
- Modules that are part of the C++ standard library are imported with angle brackets
- Local modules that you have written are included using quotes
- For example:

```
// this is a C++ std lib module
#include <iostream>

// these are local files
#include "MyModule.h"
#include "AnotherModule.h"
```

Header Files

- Local files that you import should always be headers (.h), and never source files (.cpp)
- Generally, we use header files to define all of the functions and classes in a corresponding source file
- Consider a source file called `MinMax.cpp` with the following function definitions:

```
int min(int a, int b){
  return (a < b) ? a : b;
}

int max(int a, int b){
  return (a > b) ? a : b;
}
```

Header Files

- The corresponding header file, `MinMax.h`, for this source would simply be declarations of each function:

```
int min(int a, int b);
int max(int a, int b);
```

- An include directive for this header must be added to the top of `MinMax.cpp` (otherwise the compiler will think you are redeclaring these functions)
- Now you can use `#include ``MinMax.h''` in any of your other source files to gain access to these functions
- We will see how to create header files for class definitions later

Some Frequently Used Modules

- The following are some frequently used modules from the standard library:

```
#include <iostream>     console input and output
#include <fstream>      file input and output
#include <cmath>        math functions
#include <string>       string type and functions
#include <vector>       vector container
#include <list>         list container
#include <set>          set container
#include <algorithm>    common algorithms
```

Namespaces

- C++ uses namespaces to provide different scope groups – this is done to avoid name collisions
- All of the functions and classes that are imported from the standard library using the namespace "std"
- To access a namespace, we use the scope resolution operator (::)
- For example, if we import the string module from the standard library and wanted to use the string class, we would use std::string:

```
#include <string>

std::string s = "I'm a string";
```

Namespaces

- If we can be sure that no name collisions will happen, it is convenient to just "use" a namespace instead of individually scoping everything
- We can do this with the `using` keyword:

```
#include <string>
using namespace std;

string s = "I'm a string";
```

- This is similar to "import Module" vs "from Module import *" in Python

Enumerated Types

- Enumerated types are a common programming construct where a type is defined to have a finite set of named values
- These names don't actually do anything other than provide distinction between values in a meaningful way – they are really the same as integers
- When used properly they result in much more readable and understandable code
- Think, for example, of using 0 through 6 to represent days of the week vs. using an enum with the set of values {SAT, SUN, MON, TUE, WED, THU, FRI}

Enumerated Type Example

- In C++ we can define an enumerated type using the keyword enum:

```
enum Color {RED, BLUE, YELLOW};

void colorFunc(Color c){
  if (c == BLUE)
    ...
  else if (c == RED)
    ...
  else
    ...
}
```

Classes

- Class syntax is similar to Python and Java
- To define a new class called Example, the code looks like this:

```
class Example {
  private:
    // private fields and methods go here

  protected:
    // protected fields and methods go here

  public:
    // public fields and methods go here
};
```

Class Access Specifiers

- Class members all have an associated access specifier (private, protected, or public)
- private:
  - these members are not visible outside of the class definition
  - instances cannot access these members
  - derived classes cannot access these members
- protected:
  - same as private, except derived classes can access these members
- public:
  - visible to everyone
  - class instances and derived classes can access these

Constructors

- All classes have constructors for creating instances of the class
- By default, a constructor that takes no parameters exists even if not defined – this constructor simply creates an instance
- Custom constructor declarations/definitions look like normal function declarations/definitions, except they have no return type
- Constructors must have the same name as the class
- Constructors should always be public, otherwise they can't be accessed by instances

Constructor Example

```
class Point {
  private:
    double x;
    double y;

  public:
    Point() {
      this->x = 0;
      this->y = 0;
    }
    Point(double x, double y){
      this->x = x;
      this->y = y;
    }
};
```

this

- In the example on the previous slide, we saw the keyword `this`
- `this` behaves like `self` in Python – it refers to the calling instance
- Note the arrow operator (->) – you cannot use dot (.) with `this`
- `this` is available in all class method definitions, not just the constructors

Classes in Header Files

- Similar to functions, whenever we define a class we should separate the declaration and the definition into a header file and a source file respectively
- Consider the following class definition (next slide)

Classes in Header Files

```
class Point {
  private:
    double x;
    double y;

  public:
    Point(double x, double y){
      this->x = x;
      this->y = y;
    }
    double getX(){ return this->x; }
    double getY(){ return this-y; }
};
```

Classes in Header Files

- To create a header file for this class, we remove the definitions and just give the declarations:

```
class Point {
  private:
    double x;
    double y;

  public:
    Point(double x, double y);
    double getX();
    double getY();
};
```

Classes in Header Files

- In the corresponding source file, we don't want to rewrite the class – the compiler would tell us that it is already defined
- All we would like to do is add definitions for the class methods – that is all that is missing from the header
- We can do this using the scope resolution operator:

```
#include "Point.h"

Point::Point(double x, double y){
  this->x = x;
  this->y = y;
}
double Point::getX(){ return this->x; }
double Point::getY(){ return this->y; }
```

Class Inheritance

- In C++, subclasses can be created using the following syntax:

```
class Parent {
  ...
}

// create class Child as a subclass of Parent
class Child : public Parent {
  ...
}
```

Class Inheritance

- When defining a subclass, you have access to all public and protected members of the base class
- Remember that using private members in the base class means that those members will not be accessible when writing the definitions of the subclass: it generally makes sense to use protected rather than private when you have inheritance
- Class instances of a subclass have access to all public methods and fields in the base class, as well as any additional public members defined in the subclass

Polymorphism

- Polymorphism: "having multiple forms of one thing"
- Polymorphism occurs in classes when we have different method definitions of the same method in parent classes and subclasses – this is called overriding
- Methods of the base class that will be overrided should be marked with the keyword `virtual`

```
class Animal {
  public:
    virtual void speak(){ cout << "Roar"; }
};
class Dog : public Animal {
  public:
    void speak(){ cout << "Woof"; }  // override
};
```

Generic Types

- C++ is statically typed: we must explicitly state the type of every variable in our code
- This includes function return types, function parameter types, and class member field and method types
- Sometimes we would like to use generic types so that a function or class can work with multiple different types
- In C++ we use the keyword `template` to implement generics

Generic Functions

- A generic version of the min function we defined previously would be:

```
template <class T>
T min(T a, T b){
  if (a < b)
    return a;
  return b;
}
```

- This defines T to be some generic class, and we can then use T as a type in our function definition
- The min function can now be called with any type – T is inferred based on what we pass as arguments

Generic Classes

- Generic versions of classes work the same way:

```
template <class T>
class Pair {
  private:
    T a;
    T b;

  public:
    Pair(T a, T b){
      this->a = a;
      this->b = b;
    }
    ... etc.
};
```

Instances of Generic Classes

- Unlike functions, generic types in classes are not inferred
- We must explicitly state which type:

```
// this is wrong:
Pair p(3, 3);

// this is correct:
Pair<int> p(3, 3);
```

The Standard Template Library (STL)

- The Standard Template Library (STL) is a subset of the C++ standard library
- The STL includes several generic container classes (vector, list, set, queue, deque, etc.)
- Also includes algorithms and functions that operate on the containers, as well as iterators that can be used to iterate over the containers

Memory Management

- C++ is a lower level language compared to Python or Java with respect to memory management
- Memory is not fully abstracted away
- C++ allows the programmer to allocate and deallocate memory explicitly, and to access and use the memory locations of variables

Pointers

- Pointers are variables that “point” to memory locations
- Pointers are declared similar to other variables, with a * added to the end of the type:

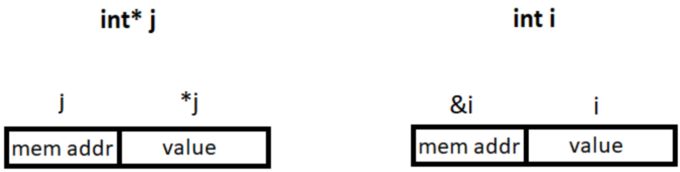
```
// this is an integer variable called i
int i;

// this is an integer pointer variable called j
int* j;

// this is an Example class pointer variable
Example* ex;
```

Referencing and Dereferencing

- Pointer variables can be dereferenced with * to access their contents
- Conversely, a reference to the memory location of a variable can be found with the & operator



Pointer Example

```
int i = 5;   // int variable i with value 5
int* j;      // int pointer variable j

j = &i;      // j points to i's memory address
*j = 7;      // change the value at address j to 7

std::cout << *j << std::endl;
std::cout << i << std::endl;

/*  this prints
      7
      7
*/
```

Allocating Memory with new

- When pointers are declared, they initially point to garbage (some random address)
- As seen in the previous slides, we can point to addresses of pre-existing variable using &
- Often when using pointers we want to allocate new memory, fill it with something, and point to that: this is done via the keyword new

```
// supposing we have a class Point
// with constructor that takes x and y values
Point* p = new Point(3,4);

// new vector from STL with default constructor
vector<int>* v = new vector<int>();
```

De-allocating Memory with `delete`

- There is no garbage collection in C++
- Whenever we allocate new memory using `new`, we need to deallocate that memory when we are done with it using the keyword `delete`:

```
Vector<int>* v = new Vector<int>();

... at some point later in code ...
// deallocates memory that was allocated to v
delete v;
```

- It is very important to remember to deallocate memory that has been allocated; failure to do so will lead to "memory leak"

A Note About Class Instances

- To access members of a class instance, we use dot (.)
- To access members of a class instance pointer, we use arrow (->) – we've seen this with the **this** keyword, which is actually a pointer in C++

```
MyClass m1 = MyClass();
MyClass* m2 = new MyClass();

// use dot to access members of m1
m1.myField;
m1.myFunction();

// use arrow to access members of m2
m2->myField;
m2->myFunction();
```

Why Use Pointers?

- You might wonder: "why would we use pointers?"
- One reason is to keep variables alive through different scopes:
  - The memory associated with variables declared in a certain scope have a lifetime which ends when that scope ends
  - Declaring a pointer and allocating memory to it with `new` will keep that memory alive until we `delete` it
- Another reason is efficiency:
  - When we call a function with parameters, the supplied parameters are copied and the copies are used locally in the function
  - This is very inefficient when we are passing large data structures
  - We could instead pass a pointer to the structure and all that needs to be copied is the integer memory address

C++ Exception Handling

- C++ has a standard library called <exception> that is used for exception handling
- Exception handling in C++ is done using a `try...catch` block:

```
try
{
  ...some code that might cause exception...
}
catch (exception& e)
{
  // handle exception here
  cout << e.what() << endl;
}
```

Custom Exceptions

- We can create custom exceptions in C++ by creating new classes that inherit from the exception class:

```
#include <exception>
using namespace std;

class MyException : public exception {
  virtual const char* what() const throw()
  {
    return "Exception message";
  }
}
```

- This exception can then be thrown in code using:

```
throw MyException();
```

C++ Example

- An example of A4 from a previous year is given in the src folder
- This example gives an MIS and corresponding C++ implementations of each module

Additional Resources

- www.cplusplus.com/ is excellent
- https://stackexchange.com/ for specific questions – very high chance your question has already been asked and answered there
- The C++ Programming Language by Bjarne Stroustrup (the creator of C++) – should be able to access via mcmaster library online access