Introduction to C++: Part 2
Tutorial Outline: Part 2

- Compiler Options
- References and Pointers
- Function Overloads
- Generic Functions
- Defining Classes
Compiler Options

- We’ve been working with Eclipse using a Debug build. This is determined by the options set for the compiler.
- The g++ compiler has a vast array of options:
- The -g flag tells the compiler to add extra information to the executable to allow the debugger to read and manipulate the program.
- Turning on optimizations makes the compiler do more work to produce code that will execute faster.
  - This is used in the Eclipse Release build.
Compiler Options (for g++ 4.8.5)

- **Common flags:**
  - `-g` Support for debugging. Sometimes not completely effective with (any) optimization turned on.
  - `-O`, `-O2`, `-O3` Produce optimized code. The higher numbers turn on more potential optimization paths for the compiler to take when generating code. They are less likely to have an impact.
  - `-ffast-math` May produce code that does not conform to IEEE standards for floating point computations. Try it with your program and see if it has any impact on accuracy and/or speed.
  - `-march=corei7` On the SCC, allow for some special CPU instructions to be generated for calculations in loops that may result in faster code.
C++ defaults to *pass by value* behavior when calling a function.

The function arguments are **copied** when used in the function.

Changing the value of L or W in the `RectangleArea1` function does **not** effect their original values in the `main()` function.

When passing objects as function arguments it is important to be aware that potentially large data structures are automatically copied!
Pass by Reference

- *Pass by reference* behavior is triggered when the `&` character is used to modify the type of the argument.
- This is the type of behavior you see in Fortran, Matlab, Python, and others.
- Pass by reference function arguments are **NOT** copied. Instead the compiler sends a *pointer* to the function that references the memory location of the original variable. The syntax of using the argument in the function does not change.
- Pass by reference arguments almost always act just like a pass by value argument when writing code except that changing their value changes the value of the original variable!!
- The `const` modifier can be used to prevent changes to the original variable in `main()`.

```c
main()
float L
float W

RectangleArea3(const float& L, const float& W)
float L
float W
```
In RectangleArea4 the pass by reference behavior is used as a way to return the result without the function returning a value.

The value of the area argument is modified in the main() routine by the function.

This can be a useful way for a function to return multiple values in the calling routine.
In C++ arguments to functions can be objects…
  - Example: Consider a string variable containing 1 million characters (approx. 1 MB of RAM).
    - Pass by value requires a copy – 1 MB, pass by reference requires 8 bytes!

Pass by value could potentially mean the accidental copying of large amounts of memory which can greatly impact program memory usage and performance.

When passing by reference, use the \textit{const} modifier whenever appropriate to protect yourself from coding errors.
  - Generally speaking – use \textit{const} anytime you don’t want to modify function arguments in a function.

“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.” – Bjarne Stroustrup
Tutorial Outline: Part 2

- Compiler Options
- References and Pointers
- Function Overloads
- Generic Functions
- Defining Classes
Function overloading

- The same function can be implemented multiple times with different arguments.
- This allows for special cases to be handled, or specialized behavior for different types.
- `cout` and the `<<` operator are an example of function overloading
  - `<<` is just a function.

```cpp
float sum(float a, float b) {
    return a + b;
}

int sum(int a, int b) {
    return a + b;
}
```
Function overloading

- Overloaded functions are differentiated by their arguments and not the return type.
  - The number of arguments and their types can be varied.

- The compiler will decide which overload to use depending on the types of the arguments.

- If it can’t decide a compile-time error will occur.

```c
float sum(float a, float b) {
    return a + b;
}

int sum(int a, int b) {
    return a + b;
}
```
C++ Templates (aka generics)

- Generic code is code that works on multiple different data types but is only coded once.
- In C++ this is called a template.

- A C++ template is implemented entirely in a header file to define generic classes and functions.
- The actual code is generated by the compiler wherever the template is used in your code.
  - There is NO PENALTY when your code is running!
  - Function overloads are created automatically by the compiler.
- As a preview of how the C++ Standard Template Library works we’ll walk thru some templates with NetBeans.
Sample template function

- The template is started with the keyword `template` and is told it'll handle a type which is referred to as `T` in the code.
  - Templates can be created with multiple different types, not limited to just one.
  - You don't have to use `T`, any non-reserved word will do.

- When the compiler sees the call to the template function it will automatically generate a function that takes and returns float types.

```cpp
template <typename T>
T sum_template (T a, T b) {
    return a+b ;
}

// Then call the function:
float x=1.0 ;
float y=2.0 ;
float z=sum_template<float>(x,y) ;
```
An Example

- Open the project *Overloads_and_templates*

- This is an example of simple function overloads and a template function.

- Let’s walk through it with the debugger.
When to use function overloading and templates?

- When it makes your code easier to use, maintain, write, or debug!
  - From an academic scientific computing point of view, that is.

- These are more advanced C++ features. Mis-use can cause a lot of misery and confusion.

- These are worthwhile parts of the language to become comfortable for more experienced C++ programmers.
Stepping back a bit

- **Summary so far:**
  - Basics of C++ syntax
  - Declaring variables
  - Defining functions
  - Using the IDE

- As an object-oriented language C++ supports a core set of OOP concepts.

- Knowing these concepts help with understanding some of the underlying design of the language and how it operates in your programs.
Object-oriented programming (OOP):
- Defines *classes* to represent data and logic in a program. Classes can contain *members* (data) and *methods* (internal functions).
- Creates *instances* of classes, aka *objects*, and builds the programs out of their interactions.

The core concepts in addition to classes and objects are:
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction
Core Concepts

- **Encapsulation**
  - Bundles related data and functions into a class

- **Inheritance**
  - Builds a relationship between classes to share class members and methods

- **Abstraction**
  - The hiding of members, methods, and implementation details inside of a class.

- **Polymorphism**
  - The application of the same code to multiple data types
Core Concepts in this tutorial

- **Encapsulation**
  - Demonstrated by writing some classes

- **Inheritance**
  - Write classes that inherit (re-use) the code from other classes

- **Abstraction**
  - Design and setup of classes, discussion of the Standard Template Library (STL)

- **Polymorphism**
  - Function overloading, template code, and the STL
Tutorial Outline: Part 2

- Compiler Options
- References and Pointers
- Function Overloads
- Generic Functions
- Defining Classes
A first C++ class

- Open project `Basic_Rectangle`.

- We’ll add our own custom class to this project.

- A C++ class consists of 2 files: a header file (.h) and a source file (.cpp)

- The header file contains the definitions for the types and names of members, methods, and how the class relates to other classes (if it does).

- The source file contains the code that implements the functionality of the class

- Sometimes there is a header file for a class but no source file.
Using Eclipse

- An IDE is very useful for setting up code that follows patterns and configuring the build system to compile them.

- This saves time and effort for the programmer.

- Right-click on the Basic_Rectangle project and choose New→Class
- Give it the name *Rectangle* and click the Finish button.

- Open the new files *Rectangle.h* and *Rectangle.cpp*
Rectangle.h

class Rectangle {
public:
    Rectangle();
    virtual ~Rectangle();
};

#ifndef RECTANGLE_H_
#define RECTANGLE_H_

/*
 * Rectangle.h
 *
 * Created on: Sep 9, 2019
 *    Author: bgregor
 */

#endif /* RECTANGLE_H_ */
Default declared methods

- Rectangle();
  - A constructor. Called when an object of this class is created.

- ~Rectangle();
  - A destructor. Called when an object of this class is removed from memory, i.e. destroyed.
  - Ignore the virtual keyword for now.
Class_name:: pattern indicates the method declared in the header is being implemented in code here.

Methods are otherwise regular functions with arguments () and matched curly braces {}.

Header file included
Let’s add some functionality

- A Rectangle class should store a length and a width.
- To make it useful, let’s have it supply an Area() method to compute its own area.
- Edit the header file to look like the code to the right.

```cpp
class Rectangle {
public:
    Rectangle();
    virtual ~Rectangle();

    float m_length;
    float m_width;

    float Area();
    float ScaledArea(const float scale);
};
```
Encapsulation

- Bundling the data and area calculation for a rectangle into a single class is an example of the concept of *encapsulation*.
The code for the two methods is needed

- Right-click in the Rectangle.h window and choose Source → Implement Methods
Click *Select All* then click OK.
Fill in the methods

- Step 1: add some comments.
- Step 2: add some code.

- **Member variables can be accessed as though they were passed to the method.**
- Methods can also call each other.
- Fill in the Area() method and then **write your own** ScaledArea(). Don’t forget to compile!
Using the new class

- Open *Basic_Rectangle.cpp*
- Add an include statement for the new Rectangle.h

- Create a Rectangle object and call its methods.
- We’ll do this together…
Special methods

- There are several methods that deal with creating and destroying objects.

- These include:
  - *Constructors* – called when an object is created. Can have many defined per class.
  - *Destructor* – one per class, called when an object is destroyed
  - *Copy* – called when an object is created by copying an existing object
  - *Move* – a feature of C++11 that is used in certain circumstances to avoid copies.
Construction and Destruction

- The constructor is called when an object is created.

- This is used to initialize an object:
  - Load values into member variables
  - Open files
  - Connect to hardware, databases, networks, etc.

- The destructor is called when an object goes *out of scope*.

- Example:

  ```
  void function() {
      ClassOne c1;
  }
  ```

- Object c1 is created when the program reaches the first line of the function, and destroyed when the program leaves the function.
When an object is instantiated...

- The rT object is created in memory.
- When it is created its constructor is called to do any necessary initialization.

- The constructor can take any number of arguments like any other function but it cannot return any values.

- What if there are multiple constructors?
  - The compiler follows standard function overload rules.

```cpp
#include "Rectangle.h"

int main(int argc, char** argv) {
    Rectangle rT;
    rT.m_width = 1.0;
    return 0;
}

Rectangle::Rectangle() {
}
```

Note the constructor has no return type!
A second constructor

Adding a second constructor is similar to overloading a function.

Here the modern C++11 style is used to set the member values – this is called a member initialization list.
Member Initialization Lists

- Syntax:

```cpp
MyClass(int A, OtherClass &B, float C):
    m_A(A),
    m_B(B),
    m_C(C) {
    /* other code can go here */
}
```

Members assigned and separated with commas. The order doesn’t matter.

Additional code can be added in the code block.
And now use both constructors

- Both constructors are now used. The new constructor initializes the values when the object is created.
- Constructors are used to:
  - Initialize members
  - Open files
  - Connect to databases
  - Etc.

```cpp
#include <iostream>
using namespace std;
#include "Rectangle.h"

int main(int argc, char** argv)
{
    Rectangle rT;
    rT.m_width = 1.0;
    rT.m_length = 2.0;
    cout << rT.Area() << endl;

    Rectangle rT_2(2.0, 2.0);
    cout << rT_2.Area() << endl;

    return 0;
}
```
Default values

- C++11 added the ability to define default values in headers in an intuitive way.

- Pre-C++11 default values would have been coded into constructors.

- If members with default values get their value set in constructor than the default value is ignored.
  - i.e. no “double setting” of the value.

```cpp
class Rectangle {
public:
    Rectangle();
    Rectangle(const float width, const float length);
    Rectangle(const Rectangle& orig);
    virtual ~Rectangle();

    float m_length = 0.0;
    float m_width = 0.0;

    float Area();
    float ScaledArea(const float scale);
};
```
Default constructors and destructors

- The two methods created by Eclipse automatically are explicit versions of the `default` C++ constructors and destructors.

- Every class has them – if you don’t define them then empty ones that do nothing will be created for you by the compiler.
  - If you really don’t want the default constructor you can delete it with the `delete` keyword.
  - Also in the header file you can use the `default` keyword if you like to be clear that you are using the default.

```cpp
class Foo {
    public:
    Foo() = delete;
    // Another constructor // must be defined!
    Foo(int x);
};

class Bar {
    public:
    Bar() = default;
};
```
Custom constructors and destructors

- You must define your own constructor when you want to initialize an object with arguments.

- A custom destructor is **always** needed when internal members in the class need special handling.
  - Examples: manually allocated memory, open files, hardware drivers, database or network connections, custom data structures, etc.
Destructors

- Destructors are called when an object is destroyed.
- Destructors have no return type.
- There is only one destructor allowed per class.
- Objects are destroyed when they go out of scope.
- Destructors are never called explicitly by the programmer. Calls to destructors are inserted automatically by the compiler.

```cpp
Rectangle::~Rectangle() {
}
```

This class just has 2 floats as members which are automatically removed from memory by the compiler.

```
~House() destructor
```

House object
Destructors

- Example:

```cpp
class Example {
    public:
        Example() = delete;
        Example(int count);

        virtual ~Example() ;

        // A pointer to some memory
        // that will be allocated.
        float *values = nullptr;
};

Example::Example(int count) {
    // Allocate memory to store "count"
    // floats.
    values = new float[count];
}

Example::~Example() {
    // The destructor must free this
    // memory. Only do so if values is not
    // null.
    if (values) {
        delete[] values ;
    }
}
```
Scope

- Scope is the region where a variable is valid.
- Constructors are called when an object is created.
- Destructors are only ever called implicitly.

```c
int main() { // Start of a code block
    // in main function scope
    float x; // No constructors for built-in types
    ClassOne c1; // c1 constructor ClassOne() is called.
    if (1) { // Start of an inner code block
        // scope of c2 is this inner code block
        ClassOne c2; // c2 constructor ClassOne() is called.
    } // c2 destructor ~ClassOne() is called.
    ClassOne c3; // c3 constructor ClassOne() is called.
} // leaving program, call destructors for c3 and c1 ~ClassOne()
// variable x: no destructor for built-in type
```
Copy, Assignment, and Move Constructors

- The compiler will automatically create constructors to deal with copying, assignment, and moving. NetBeans filled in an empty default copy constructor for us.

- How do you know if you need to write one?
  - When the code won’t compile and the error message says you need one!
  - OR unexpected things happen when running.

- You may require custom code when...
  - dealing with open files inside an object
  - The class manually allocated memory
  - Hardware resources (a serial port) opened inside an object
  - Etc.

```java
Rectangle rT_1(1.0, 2.0); // Now use the copy constructor
Rectangle rT_2(rT_1); // Do an assignment, with the
// default assignment operator
rT_2 = rT_1;
```
Templates and classes

- Classes can also be created via templates in C++

- Templates can be used for type definitions with:
  - Entire class definitions
  - Members of the class
  - Methods of the class

- Templates can be used with class inheritance as well.

- This topic is way beyond the scope of this tutorial!