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

- References and Pointers
- Function Overloads
- Generic Functions
- Defining Classes
- Intro to the Standard Template Library
Pass by Value

- 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().

```
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 const modifier whenever appropriate to protect yourself from coding errors.

- Generally speaking – use 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

- References and Pointers
- Function Overloads
- Generic Functions
- Defining Classes
- Intro to the Standard Template Library
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.
The formal concepts in OOP

- **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

- References and Pointers
- Function Overloads
- Generic Functions
- Defining Classes
- Intro to the Standard Template Library
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 NetBeans

- 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→C++ Class
- Give it the name *Rectangle* and click the Finish button.

- Under the *Header Files* in the project open the new *Rectangle.h* file.
Rectangle.h

```cpp
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle {

public:
    Rectangle();
    Rectangle(const Rectangle& orig);
    virtual ~Rectangle();

private:

};

#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.

- **Rectangle(const Rectangle& orig);**
  - A *copy* constructor. Used to create a new object that’s a copy of another.

```cpp
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle {
public:
    Rectangle();
    Rectangle(const Rectangle& orig);
    virtual ~Rectangle();

private:
};
#endif /* RECTANGLE_H */
```
Rectangle.cpp

```cpp
#include "Rectangle.h"

Rectangle::Rectangle() {
}

Rectangle::Rectangle(const Rectangle& orig) {
}

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

**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 {}.
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();
    Rectangle(const Rectangle& orig);
    virtual ~Rectangle();

    float m_length ;
    float m_width ;

    float Area() ;
    float ScaledArea(const float scale);

private:
};
```
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

- Click on Rectangle.cpp and put the cursor at the end of the file.
- Type Ctrl-Space
- Select the Area() method.
- Repeat for ScaledArea().
- This creates a stub with necessary stuff filled in.
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 `main.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:

  ```c
  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()
{
    Rectangle rT;
    rT.m_width = 1.0;
}
```

```cpp
#include "rectangle.h"

Rectangle::Rectangle()
{
    //ctor
}
```

Note the constructor has no return type!
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.

Colon goes here

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>
#include "rectangle.h"

using namespace std;

int main()
{
    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);

private:
};
```
Default constructors and destructors

- The two methods created by NetBeans 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.

```
Rectangle::~Rectangle()
{
    //dtor
}
```

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.

```cpp
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!
Tutorial Outline: Part 2

- References and Pointers
- Function Overloads
- Generic Functions
- Defining Classes
- Intro to the Standard Template Library
The Standard Template Library

- The STL is a large collection of containers and algorithms that are part of C++.
  - It provides many of the basic algorithms and data structures used in computer science.

- As the name implies, it consists of generic code that you specialize as needed.

- The STL is:
  - Well-vetted and tested.
  - Well-documented with lots of resources available for help.
Containers

- There are 16 types of containers in the STL:

<table>
<thead>
<tr>
<th>Container</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>1D list of elements.</td>
</tr>
<tr>
<td>vector</td>
<td>1D list of elements</td>
</tr>
<tr>
<td>deque</td>
<td>Double ended queue</td>
</tr>
<tr>
<td>forward_list</td>
<td>Linked list</td>
</tr>
<tr>
<td>list</td>
<td>Double-linked list</td>
</tr>
<tr>
<td>stack</td>
<td>Last-in, first-out list.</td>
</tr>
<tr>
<td>queue</td>
<td>First-in, first-out list.</td>
</tr>
<tr>
<td>priority_queue</td>
<td>1st element is always the largest in the container</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Container</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>Unique collection in a specific order</td>
</tr>
<tr>
<td>multiset</td>
<td>Elements stored in a specific order, can have duplicates.</td>
</tr>
<tr>
<td>map</td>
<td>Key-value storage in a specific order</td>
</tr>
<tr>
<td>multimap</td>
<td>Like a map but values can have the same key.</td>
</tr>
<tr>
<td>unordered_set</td>
<td>Same as set, sans ordering</td>
</tr>
<tr>
<td>unordered_multiset</td>
<td>Same as multiset, sans ordering</td>
</tr>
<tr>
<td>unordered_map</td>
<td>Same as map, sans ordering</td>
</tr>
<tr>
<td>unordered_multimap</td>
<td>Same as multimap, sans ordering</td>
</tr>
</tbody>
</table>
Algorithms

- There are 85+ of these.
  - Example: find, count, replace, sort, is_sorted, max, min, binary_search, reverse
- Algorithms manipulate the data stored in containers but is not tied to STL containers
  - These can be applied to your own collections or containers of data
- Example:

  ```cpp
  vector<int> v(3); // Declare a vector of 3 elements.
  v[0] = 7;
  v[1] = 3;
  ```

- The implementation is hidden and the necessary code for reverse() is generated from templates at compile time.
vector<T>

- A very common and useful class in C++ is the vector class. Access it with:

```cpp
#include <vector>
```

- Vector has many methods:
  - Various constructors
  - Ways to iterate or loop through its contents
  - Copy or assign to another vector
  - Query vector for the number of elements it contains or its backing storage size.

- Example usage: `vector<float> my_vec;`
- Or: `vector<float> my_vec(50);`
vector<T>

- Hidden from the programmer is the **backing store**
- Object oriented design in action!

- This is how the vector stores its data internally.

Contains N elements. Given by size() method.

Allocated for a total of M elements
  Given by capacity() method.

Add some more to the vector

Old allocation is destroyed.

Allocated for a total of M' elements

New memory is allocated.
Old data is copied in.
New M > old M.
Destructors

- `vector<t>` can hold objects of any type:
  - Primitive (aka basic) types: `int`, `float`, `char`, etc.
  - Objects: `string`, your own classes, file stream objects (ex. `ostream`), etc.
  - Pointers: `int*`, `string*`, etc.
  - But NOT references!

- **When a vector is destroyed:**
  - If it holds primitive types or pointers it just deallocates its backing store.
  - If it holds objects it will call each object’s destructor before freeing its backing store.
vector<t> with objects

- Select an object in a vector.
- The members and methods can be accessed directly.
- Elements can be accessed with brackets and an integer starting from 0.

```cpp
// a vector with memory preallocated to hold 1000 objects.
vector<MyClass> my_vec(1000);

// Now make a vector with 1000 MyClass objects
// that are initialized using the MyClass constructor
vector<MyClass> my_vec2(1000, MyClass(arg1, arg2));

// Access an object's method.
my_vec2[100].some_method();
// Or a member
my_vec2[10].member_integer = 100;

// Clear out the entire vector
my_vec2.clear()
// but that might not re-set the backing store...
// Let’s check the docs:
```
Loop with a “for” loop, referencing the value of vec using brackets.

1\textsuperscript{st} time through:
- index = 0
- Print value at vec[0]
- index gets incremented by 1

2\textsuperscript{nd} time through:
- Index = 1
- Etc

After last time through
- Index now equal to vec.size()
- Loop exits

Careful! Using an out of range index will likely cause a memory error that crashes your program.

Note we call the size() method on every iteration.
Iterators are generalized ways of keeping track of positions in a container.  
- 3 types: forward iterators, bidirectional, random access  
- Forward iterators can only be incremented (as seen here)  
- Bidirectional can be added or subtracted to move both directions  
- Random access can be used to access the container at any location
### Looping

- Loop with a “for” loop, referencing the value of vec using an iterator type.

  - `vector<int>::iterator` is a type that iterates through a vector of int’s.

- **1st time through:**
  - itr points at 1st element in vec
  - Print value pointed at by itr: `*itr`
  - itr is incremented to the next element in the vector

- Iterators are very useful C++ concepts. They work on any STL container!
  - **No need to worry about the # of elements!**
  - Exact iterator behavior depends on the type of container but they are guaranteed to always reach every value.

- Note we are now retrieving the end iterator at every loop to see if we’ve reached it: `vec.end()`
Let the \textit{auto} type ask the C++ compiler to figure out the iterator type automatically.

An extra modification: Assigning the vec\_end variable avoids calling vec.end() on every loop.
Another iterator-based loop: iterator behavior and accessing an element are handled automatically by the compiler.

- Uses a reference so the element is not copied.
- The `const auto &` prevents changes to the element in the vector.
- Less typing == less chance for program bugs.

```cpp
for(const auto &element : vec) {
    cout << element << " " ;
}
```
Iterator notes

- There is small performance penalty for using iterators…but are they safer to use.
- They allow substitution of one container for another (list<> for vector<>, etc.)
- With templates you can write a function that accepts any STL container type.

```cpp
template<class T>
void dump_string(T &t)
{
    for( auto itr=t.begin() ; itr!=t.end() ; itr++ ) {
        cout<<*itr<<endl;
    }
}
```

```cpp
list<float> lst;
lst.push_back(-5.0);
lst.push_back(12.0);
vector<double> vec(2);
vec[0] = 1.0;
vec[1] = 2.0;

dump_string<list<float>>(lst);
dump_string<vector<double>>(lst);
```
STL Demo

- Open project *STL_Demo*

- Let’s walk through the functions with the debugger and see some vectors in action.