

# C++ for A3

(miscellaneous topics that will be helpful for A3)

## CS 2ME3/SE 2AA4

Steven Palmer

Department of Computing and Software  
McMaster University

# Outline

- 1 Compiling on Mills
- 2 Initializer Lists
- 3 Template Classes
- 4 typedef
- 5 Midterm Review

## A Note About Compiling on Mills

- Your code for A3 should adhere to the C++11 standard
- The default version of g++ on mills is outdated – it will not recognize the `-std=c++11` option
- If you try to run the makefile for A3 on Mills, it will fail
- You must first run the command:  
    `. /opt/rh/devtoolset-7/enable`
- This will update the g++ version to one that supports C++11
- You will then be able to compile A3 successfully

# Automating It

- Everytime you ssh into Mills, you will need to run the command on the previous slide
- For convenience, you can edit your `.bashrc` file so that this command is automatically run every time you log in by doing the following:

1 run

```
nano ~/.bashrc
```

2 paste

```
. /opt/rh/devtoolset-7/enable
```

on a new line at the end of the file

3 CTRL + O, then ENTER to save

4 CTRL + X to exit nano

# Initializing Classes

- When you create a class instance in C++, all class fields are initialized to default values *before* the constructor is called
- This can be a problem if you've defined classes for which you have supplied custom constructors, but have not defined a default constructor (one that takes no arguments)
- For an example of why this might be a problem, see the example code on the slide

## A Failing Example

```
class MyClass {  
    private:  
        int a;  
    public:  
        MyClass(int a) { this->a = a; }  
};  
  
class MyOtherClass {  
    private:  
        MyClass m;  
    public:  
        MyOtherClass(MyClass mc) { this->m = mc; }  
};
```

## Why The Previous Example Fails

- The code on the previous slide will fail to compile
- In `MyClass`, we did not define a default constructor – this class has no default value
- Since the fields of class instances are initialized to default values before the constructor is called, the compiler won't know what to do about the `MyClass m` field of `MyOtherClass`
- Even though we never actually use the default value, and we assign to `m` in the constructor, the compiler will still insist on a default value
- This happens fairly often, where we have a class with no default constructor as a field of another class – so how do we get around this problem?

# Initializer Lists

- Initializer lists are used to initialize class variables immediately with values when the class is being instantiated – no default values are constructed for fields with initializers
- When defining a constructor, we can put an initializer list right after the signature like this:

```
Class::Class(...) : fld1(val1), fld2(val2), ...  
{  
    // constructor body if necessary  
    // or just blank body  
}
```

- where fld1, fld2, etc. are the field names that you want to initialize, and val1, val2, etc. are the values you want to initialize them with



## Initializer List Example

- For example, we could fix the previous failing example by changing the definition of the `MyOtherClass` to be:

```
class MyOtherClass {  
    private:  
        MyClass m;  
    public:  
        MyOtherClass(MyClass mc) : m(mc) { }  
};
```

- Using the initializer list means the compiler will no longer try to construct a default value for `m` when an instance of the class is created

## Exercise 1: Initializer Lists

### Exercise 1

There is some example code in `example/initializer/without`. In this code, `ClassB` has a field of type `ClassA`. In `ClassA` we have defined a constructor that takes an `int`, but no default constructor. Try compiling this code:

```
g++ -c *.cpp
```

You should find that there is a compilation error. Since we didn't use an initializer list, the compiler doesn't know how to construct a default value for the `ClassA` field in `ClassB`.

## Exercise 1: Initializer Lists

### Exercise 1

Now look at the code in `example/initializer/with`. This code is the same as before, except we have used an initializer list in `ClassB`. You should be able to compile this without issue:

```
g++ -c *.cpp
```

# Initializer List Hint

## HINT for A3:

- You will require an initializer list in your definition of the LineT constructor in LineADT.cpp

# Template (Generic) Class Implementation

- Template classes are not actually class definitions – they are patterns that the compiler uses to generate a family of classes
- To generate a class from a template class, the compiler needs to “see” the entire pattern – it needs the full definition, not just the declaration
- If we compile .cpp files without the full definition, the linker will complain about undefined references
- This means that if we separate the declaration and definitions of a template class into .h and .cpp files, we will run into linking errors when trying to link other source files that use instantiations of the template class

# Ways to Handle Template Class Issue

There are two ways around this issue, but each come with pros and cons:

- 1 Explicit instantiation of the template class with particular types in the .cpp file
  - Pro: you are still able to hide the implementation
  - Con: you can only use the generic class with the types that have been explicitly instantiated
- 2 Writing the full class definition in the .h file with no .cpp file
  - Pro: you get a true generic class that can be instantiated with any type
  - Con: you expose the implementation details

# Explicit Instantiation of Template Classes

- Say we have a template class called `MyClass` that we declare in a header file, i.e.

```
template <class T>
class MyClass {
    ...
};
```

- In the corresponding source file, after we define all of the member functions, we can make explicit instantiations via:

```
template class MyClass<int>;
template class MyClass<bool>;
template class MyClass<MyOtherClass>;    // etc.
```

## Exercise 2: Explicit Instantiation of Template Classes

### Exercise 2

There is some example code in `example/template`. This code declares a template class called `Example` in `Example.h`, with definitions in `Example.cpp`. There is also a main function in `main.cpp` that tries to make instances of the `Example` class instantiated with `int` and `double`.

Try compiling this code into a program:

```
g++ -c *.cpp
g++ -o prog *.o
```



## Exercise 2: Explicit Instantiation of Template Classes

### Exercise 2

Your compilation attempt should have failed with a linking error about undefined references to `Example<int>` and `Example<double>`. The linker sees we are trying to use `Example<int>` and `Example<double>` in `main.o`, but can't resolve them to a type. This is a result of the problem discussed earlier – we can't separate the declaration and definition of a template class in the usual way.

Now let's try adding explicit instantiations. At the bottom of `Example.cpp`, add the following lines:

```
template class Example<int>;  
template class Example<double>;
```

## Exercise 2: Explicit Instantiation of Template Classes

### Exercise 2

Now try compiling again:

```
g++ -c *.cpp  
g++ -o prog *.o
```

The explicit instantiations of `Example<int>` and `Example<double>` cause the compiler to store the full definition of those classes in `Example.o`. Now when linker can resolve the types referenced in `main.o` and compilation is successful.

## Template Classes Hint

- In A3, you need to implement the generic class Seq2D<T>
- Since you know ahead of time that you will only need two type instances of the Seq2D class (LanduseT and int), **you should use explicit instantiation**
- **HINT for A3:** the last two lines of your Seq2D.cpp file should be:

```
template class Seq2D <LanduseT>;  
template class Seq2D <int>;
```

# The typedef Keyword

- typedef is used to create type aliases in C++

- Usage:

```
typedef <known type> <new type alias>
```

- For example:

```
// nat as alias for unsigned int
typedef unsigned int nat;
// real as alias for double
typedef double real;
// myIntClass as alias for MyClass<int>
typedef MyClass<int> myIntClass;
```

## Using typedef'd Types

- Once typedef's have been made, you can use them like any other type
- For example, with the typedef's from the previous slide, we could then write:

```
nat n = 0;  
real r = 1.50;
```

```
// assuming MyClass(T) constructor exists:  
myIntClass mic(5);
```

## typedef Hint

### HINT for A3:

- The LanduseMap and DEM modules will just be typedef's in the LanduseMap.h and DEM.h files
- These module don't need .cpp files – no further implementation is required (the implementation is all done in Seq2D.cpp!)

# Midterm Review

We will now take up the midterm.