CMSC202
Computer Science II for Majors

Lecture 17 and 18 –
Bits & Pieces and Templates

Dr. Katherine Gibson

Last Class We Covered

- Error handling
- Exceptions
  - Try
  - Throw
  - Catch

- Went over Exam 2

Any Questions from Last Time?

Today's Objectives

- Bits & Pieces
  - Initialization lists
  - The "grep" command
  - Redirecting input and output
- Templates
  - How to implement them
  - Possible problems (and solutions)
  - Compiling with templates

Bits & Pieces

Initialization Lists

- Initialization lists are the only way you can call a base class constructor from a derived class

```
Derived(int arg1, string arg2, float arg3)
: Base(arg1, arg2)
{
    m_derivedOnlyArg = arg3;
}
```

Must use a colon, and must come before the { } braces

The "grep" Command

- Used to search through text (*e.g.*, your code)

```
grep <flags> "text" <files to search>
```

- Useful flags (optional):
  - **`-n`** show the line number of the match
  - **`-i`** make the search case insensitive
  - **`-B #`** show # lines before the match
  - **`-C #`** show # lines after the match

The "grep" Command

- Used to search text (*e.g.*, your code)

```
grep <flags> "text" <files to search>
```

- Useful ways to search files:
  - `-*` search all files
  - `-*.cpp` search all files that end in ".cpp"
  - `-test.cpp` search only the file "test.cpp"

- You can also use the "`*`" in your search query

The "grep" Command

- Here are some example uses of grep:

```
grep -nB 3 -C 2 "int" temp.cpp
```

  – Looks for the word "int" in the temp.cpp file, and displays 3 lines before, 2 lines after, and the line #

```
grep -in "cruno*" Cruno*.cpp
```

  – Will look for any instance of the word "cruno" (upper or lower), in all of the .cpp files that start with the word "Cruno"; it shows the line # as well

Redirecting Input and Output

- Rather than printing output to the screen, we can save it in a file, using redirection
  - BTW, this has to do with GL/Linux, not C++

- Use the angle brackets (< and >) to redirect
  - Output files don't need to exist beforehand
    - The system will create one for you
  - Input files do need to exist beforehand

Redirection Examples

- Save a.out's output (*i.e.*, **`cout`**) into "output.txt"

```
./a.out > output.txt
```

- Use "input.txt" in lieu of user input for Proj7

```
./Proj7 < input.txt
```

- Use "in.txt" and save "out.txt" at the same time

```
./a.out > out.txt < in.txt
```

- Save all the output, including errors (*e.g.*, **`cerr`**)

```
./a.out >& allOutput.txt
```

Templates

Overloading Swap Function

- Here is a function to swap two integers:

```
void SwapVals(int &v1, int &v2)  {
  int temp;

  temp = v1;
  v1 = v2;
  v2 = temp;
}
```

what if we want to swap two floats?

what do we need to change?

Overloading Swap Function

- Here is a function to swap two floats:

```
void SwapVals(float &v1, float &v2) {
  float temp;

  temp = v1;
  v1 = v2;
  v2 = temp;
}
```

what if we want to swap two chars?

what do we need to change?

Overloading Swap Function

- Here is a function to swap two chars:

```
void SwapVals(char &v1, char &v2) {
  char temp;

  temp = v1;
  v1 = v2;
  v2 = temp;
}
```

what if we want to swap two strings?

what do we need to change?

Overloading Swap Function

- This is getting ridiculous!
- We should be able to write just one function that can handle all of these things
  - The only difference is the data type, after all
- This is possible by using templates

What Are Templates?

- Templates let us create functions and classes that can use "generic" input and types

- This means that functions like **`SwapVals()`** only need to be written once
  - And can then be used for almost anything

Overloaded Example

```
float    maxx ( const float a, const float b );
int      maxx ( const int a, const int b );
Rational maxx ( const Rational& a, const Rational& b);
myType   maxx ( const myType& a, const myType& b);
```

- Code for each looks the same…

```
if ( a < b )
  return b;
else
  return a;
```

We want to reuse this code for **all** types

Indicating Templates

- To let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

this keyword tells the compiler that what follows this will be a template

Indicating Templates

- To let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

this **does not** mean "class" in the same sense as C++ classes with members!

in fact, another keyword we can use is actually "**`typename`**", because we are defining a new type

but "**`class`**" is more common by far, and so we will use "**`class`**" to avoid confusion

Indicating Templates

- To let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

"**T**" is the name of our new type

we can call it anything we want, but using "**T**" is the style convention

(of course, we can't use "**int**" or "**for**" or any other types or keywords as a name for our type)

Indicating Templates

- To let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

- What this line means overall is that we plan to use "**T**" in place of a data type
  - *e.g.*, **int**, **char**, **myClass**, etc.
- This template prefix needs to be used before function declarations and function definitions

Template Example

- Function Template

```
template <class T>
T maxx ( const T& a, const T& b)
{
  if ( a < b )
    return b;
  else
    return a;
}
```

- Compiler generates code based on the argument type

```
cout << maxx(4, 7) << endl;
```

- Generates the following:

```
int maxx ( const int& a, const int& b)
{
  if ( a < b )
    return b;
  else
    return a;
}
```

Template Example

- Function Template

```
template <class T>
T maxx ( const T& a, const T& b)
{
  if ( a < b )
    return b;
  else
    return a;
}
```

Notice how '**T**' is mapped to '**int**' everywhere in the function…

- Compiler generates code based on the argument type

```
cout << maxx(4, 7) << endl;
```

- Generates the following:

```
int maxx ( const int& a, const int& b)
{
  if ( a < b )
    return b;
  else
    return a;
}
```

Using Templates

- When we call these templated functions, nothing looks different:

```
SwapVals(intOne,    intTwo);
SwapVals(charOne,   charTwo);
SwapVals(strOne,    strTwo);
SwapVals(myClassA,  myClassB);
```

(In)valid Use of Templates

- Which of the following function calls will work?

```
SwapVals(bigInt, littleInt);
SwapVals(myChar, myString);
SwapVals("hello", "world");
SwapVals(doubleVar, floatVar);
SwapVals(Shape1, Shape2);
```

These use two different types, and the SwapVals() function doesn't allow this.

These are two string literals – we can't swap those!

Template Requirements

- Templated functions can handle any input type that "makes sense"
  - *i.e.*, any data type where the behavior that occurs in the function is defined

- Even user-defined types!
  - **As long as the behavior is defined**
  - What happens if the behavior isn't defined?
    - Compiler will give you an error (maybe)
    - Your program compiles, but doesn't work right

Project 4 Announcement

- There were some questions about this, so…
- A player who has to Draw Two does NOT skip their turn!
- They can play a card after drawing two

Overloading Templates

Why Overload Templates?

- Sometimes, even though the behavior is defined, the function performs incorrectly

- Assume the code:

```
char* s1 = "hello";
char* s2 = "goodbye";
cout << maxx( s1, s2 );
```

- What is the call to **`maxx()`** actually going to do?

Incorrect Template Performance

- The compiler generates:

```
char* maxx (const char*& a, const char*& b)
{
  if ( a < b )
    return b;
  else
    return a;
}
```

- Is this what we want?
  - It's going to sort them by their address in memory!

Overloading a Template

- Fix this by creating a version of **`maxx()`** specifically to handle **`char*`** variables
  - Compiler will use this instead of the template

```
char* maxx(char *a, char *b)
{
  if (strcmp(a, b) < 0)
    return b;
  else
    return a;
}
```

Compiling Templates

Compiler Handling of Templates

- Exactly what versions of **`SwapVals()`** are created is determined at compile time

- If we call **`SwapVals()`** with integers and strings, the compiler will create versions of the function that take in integers and strings

Separate Compilation Problem

- Which versions of templated function to create are determined at compile time
- How does this affect our use of separate compilation?
  - Function declaration in `.h` file
  - Function definition in `.cpp` file
  - Function call in separate `.cpp` file

Separate Compilation: Example

- Here's an illustrative example:

```
#include "swap.h"

int main()
{
  int a = 3, b = 8;
  SwapVals(a, b);
}
```

main.cpp

```
template <class T>
void SwapVals(T &v1, T &v2);
```

swap.h

```
#include "swap.h"

template <class T>
void SwapVals(T &v1, T &v2)
{
  T temp;
  temp = v1;
  v1   = v2;
  v2   = temp;
}
```

swap.cpp

Separate Compilation Problem

- Most compilers (including GL's) cannot handle separate compilation with templates

- When **`swap.cpp`** is compiled…
  - There are no calls to **`SwapVals()`**
  - so **`swap.o`** has no **`SwapVals()`** definitions

Separate Compilation Problem

- When **`main.cpp`** is compiled…
  - It assumes everything is fine
  - Since **`swap.h`** has the appropriate declaration

- When **`main.o`** and **`swap.o`** are linked…
  - Everything goes wrong
  - **`error: undefined reference to 'void SwapVals<int>(int&, int&)'`**

Separate Compilation Solutions

- The template function definition code must be in the same file as the function call code

- Two ways to do this:
  - Place function definition in **`main.c`**
  - Place function definition in **`swap.h`**, which is **`#include`**'d in **`main.c`**

Template Compilation Solution

- Second option keeps some sense of separate compilation, and better allows code reuse

```
#include "swap.h"

int main()
{
  int a = 3, b = 8;
  SwapVals(a, b);
}
```

main.cpp

```
// declaration
template <class T>
void SwapVals(T &v1, T &v2);

// definition
template <class T>
void SwapVals(T &v1, T &v2)
{
  T temp;
  temp = v1;
  v1   = v2;
  v2   = temp;
}
```

swap.h

Class Templates

Templating Classes

- Want to be able to define classes that work with various types of objects
- Shouldn't matter what kind of object it stores

- Generic "collections" of objects
  - Linked List
  - Stack
  - Vector
  - Binary Tree (341)
  - Hash Table (341)

Making a Templated Class

- Three key steps:

1. Add template line
   – Before class declaration
2. Add template line
   – Before each method in implementation
3. Change class name to include template
   – Add **<T>** after the class name wherever it appears

Example: Templated Node

```
template <class T>
class Node
{
   public:
      Node( const T& data );
      const T& GetData();
      void SetData( const T& data );
      Node<T>* GetNext();
      void SetNext( Node<T>* next );

   private:
      T m_data;
      Node<T>* m_next;
};
```

```
template <class T>
Node<T>::Node( const T& data )
{
   m_data = data;
   m_next = NULL;
}
```

```
template <class T>
const T& Node<T>::GetData()
{
   return m_data;
}
```

```
template <class T>
void Node<T>::SetData( const T& data )
{
   m_data = data;
}
```

```
template <class T>
Node<T>* Node<T>::GetNext()
{
   return m_next;
}
```

```
template <class T>
void Node<T>::SetNext( Node<T>* next )
{
   m_next = next;
}
```

Templates as Parameters

- Not much different from a “regular” variable

```
template <class T>
void Sort ( SmartArray<T>& theArray )
{
   // code here
}
```

- Make sure that the behaviors used in the function are defined for the type you’re using

What Use are Templates?

- The STL is essentially templates on steroids
  - Standard Template Library
- Works with many custom created objects but **only** if you overload the needed operators
  - `=`, `!=`, `<`, **`compare`** (used for sorting), etc.

- Likely you will also want to overload streams
  - **`cout <<`**
  - **`cin >>`**