From Last Time: What is a Template?

- This is the “official” specification for a template.
- It says that to define a template you must:
  - start with the keyword `template`
  - specify, in angle brackets, a placeholder name for each type you are using (these aren’t necessarily classes, that’s just the name chosen for this task)
  - follow the angle brackets with your function name
  - follow the function name with your parameter list (using any combination of real types and placeholder names)
- Again, the fact that you are using the keyword `class` for each placeholder you are defining has nothing to do with a C++ class.
- At compilation time the compiler will look at your code and generate a separate function for each type used throughout your code when calling template functions.

What is a Template?

- Well, this is really the definition of a template function.
- C++ also has the notion of a template class.
- A template class in its simplest form is a normal class definition with one or more variable types used throughout this class.
- The variable types are represented by the same placeholders we saw in template functions.
- The “declaration” of the placeholders is done, again, with the `template` keyword followed by the placeholders in angled brackets.
- This “declaration” appears just before the `class` keyword.

```
template <class placeholder> // declare placeholders
class SimpleClass // regular class definition
{
    public:
        // members
    }
```

Template Classes

- As you can see, `storageType` is used throughout the class as a placeholder for a type “to be determined”
- But how do we declare an instance of this class?

```
template <class storageType>
class MyArray {
    public:
        MyArray();
        ~MyArray();
        storageType operator[](int i);
        bool setArrayBounds(int argArraySize);
    private:
        bool growStorage(int argArraySize);
        storageType *arrayStore;
        int arraySize;
};
```

- Remember how template functions could be called with a specific type to coerce the placeholder argument to?
- Declaring an instance of a template class is similar.
- You must specify a type at declaration time:

```
int main()
{
    MyArray<int> intArray; // Declares an instance of MyArray with int as storage type.
    intArray[0] = 1;
    intArray[1] = 2;
    intArray[2] = 3;
    intArray[3] = 4;
    intArray[4] = 5;
    return 0;
}
```
Template Classes

- But, we're getting a little ahead of ourselves.
- We need to provide definitions for our member functions.
- Each member function can be thought of as a template function and is defined like this:

```cpp
template <class storageType>
bool MyArray<storageType>::setArrayBounds(int i)
{
    if (i > arraySize)
    {
        return growStorage(i);
    }
    return true;
}
```

- You see, in “parameterizing” the class with the placeholder for a type “to be defined” we’re actually allowing for an infinite number of classes to be created at compile time.
- That is why all of template keywords are necessary on the member functions.
- A separate class (and thus a separate set of member functions) is generated for each different type used to create an instance of this class.
- Constructors and Destructors have a template format as well:

```cpp
template <class storageType>
MyArray<storageType>::MyArray()
{
    arrayStore = NULL;
    arraySize = 0;
}
```

- Where do you think these template member functions need to go in our source code?
- It would be natural to assume that they would go in a file called MyArray.cpp, whereas the template class definition would go in MyArray.h
- This won’t work :-) .
- Well, even though this looks like a definition it is actually more of a declaration.

Template Classes

Along with our “generic” array type called MyArray, another natural candidate for a template class is the List class.
- Consider a generic list class which lets the type of container used as “storage” in the list be abstracted through the use of templates...

Demonstration #1

Better List
But Wait, It Gets Stranger...

- When creating template classes we’ve only considered that we can specify a data type in any given placeholder.
- You can also specify a constant expression except for floating point values.
- Consider the following modifications to MyArray which impose bounds restrictions on the array size:

```
template <class storageType, int size>
class MyArray {
public:
  ~MyArray();
  -MyArray();
  storageType& operator[](int i);
private:
  storageType arrayStore[size];
  int arraySize;
};
```

- Note the new piece of syntax up by the template keyword!
- This specifies that there is an integer value to be propagated through the class as part of the type.
- It’s conceptually equivalent to defining a separate class for each value of size specified with a const member variable = size.

- One of the advantages here is that I don’t need to check the validity of size.
- If the programmer (most likely myself) passes something totally inappropriate for size (like 0 or a negative number) the compiler will catch it (since these templates are being expanded out at compile time).

But Wait, It Gets Stranger...

- An example of using a constant expression in a template placeholder is enforcing a limit on the size of a linked list.
- This isn’t as appropriate as the example just given for MyArray, because you could easily define a member function in your List class which sets the limit as well.
- It’s still interesting to look at, so...

Demonstration #2

Limited List

Now How Much Would You Pay?

- But wait, there’s more!
- How about template classes (with a designated type) as types passed to other templates?
- How about a list of Lists?

```
int main()
{
  // Declare an list of MyStrings
  List<MyString> stringList;
  MyString me = "This is weird";
  stringList.insert(me, 0);
  List < List < MyString > > myList;  // <--- COOL!
  myList.insert(stringList, 0);
  ...
}
```
Now How Much Would You Pay?

List<MyArray<MyString>> myList;
List<MyArray<int>> myArrayList;
MyArray<List<MyArray<int>>> myArrayOfListsofArray;
// You get the idea!

- And you thought function templates were strange!
- Yes, you can nest these types to an infinite depth with the usual exceptions of available system resources and sanity.
- There is one important point here. Notice the deliberate spaces between angle brackets on the right side of the declarations.
- If you defined these like this:
  - List<MyArray<MyString>>
- The compiler would think the two angle brackets on the right were the >> operator.
- This is one case in the C++ compiler where whitespace matters.

Act Now, and I’ll Throw In Inheritance!

- Just call our toll-free number 1-8xx-INHERIT
- You might ask, how can I create a single template for one of these classes built by nesting templates?
- By using what LNG calls a “degenerate” form of inheritance.
- Instead of using List<List<int>> everywhere, I can use:

```cpp
template <class storageType>
class ListofLists : public List<List<storageType>> {};  // You get the idea!

void func()
{
    // The following creates a list of lists of integers
    ListofLists<int> listOfListsOfInt;
    ...
}
```

Demonstration #3

And, I’ll throw in a FREE demo!
ListOfLists

Lecture 18

Final Thoughts