Lecture 17

Templates, Part I

What is a Template?
- In short, it's a way to define generic functionality on parameters without needing to declare their types.
- Consider the following example...
- Suppose you are in need of sorting arrays of various types.
- You may initially know that you only need to sort integers, but what if you need to sort other types down the road?
- No matter what type(s) are involved, all sorting algorithms have a common need to swap (exchange) data elements in an array.
- We could abstractly represent this with the following pseudo-code:

```
void swap(<sometype> *a, int i, int j)
{
    <sometype> temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

What is a Template?
- This pseudo-code gives us a “blueprint” for implementing the swap function for any given type.
- So, if I needed to sort arrays of integers, floating points and strings, I could provide three overloaded definitions of swap().

```
void swap(int *a, int i, int j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
void swap(float *a, int i, int j)
{
    float temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
void swap(string *a, int i, int j)
{
    string temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

What is a Template?
- But wait... being big believers in choice and our natural tendencies to like to compare things...
- We want to have sorting routines for multiple sorting methods!
- We need more prototypes (well, and function definitions too!)

```
// Insertion Sort function prototypes...
void insertionSort(int *intArray, int size);
void insertionSort(float *floatArray, int size);
void insertionSort(string *stringArray, int size);
// Selection Sort function prototypes...
void selectionSort(int *intArray, int size);
void selectionSort(float *floatArray, int size);
void selectionSort(string *stringArray, int size);
// Bubble Sort function prototypes...
void bubbleSort(int *intArray, int size);
void bubbleSort(float *floatArray, int size);
void bubbleSort(string *stringArray, int size);
```

What is a Template?
- Uh oh, I just remembered.
- We can get a better range of numbers if we use doubles instead of floats, and in past semesters I've liked using MyString better than string. So...

```
// swap function prototypes...
void swap(int *a, int i, int j);
void swap(float *a, int i, int j);
void swap(MyString *a, int i, int j);
// Insertion Sort function prototypes...
void insertionSort(int *intArray, int size);
void insertionSort(float *floatArray, int size);
void insertionSort(MyString *coolArray, int size);
```
What is a Template?

// Bubble Sort function prototypes...
void bubbleSort(int *intArray, int size);
void bubbleSort(float *floatArray, int size);
void bubbleSort(string *stringArray, int size);
// Selection Sort function prototypes...
void selectionSort(int *intArray, int size);
void selectionSort(double *doubleArray, int size);
void selectionSort(MyString *coolArray, int size);

- Had enough yet?
- Let's go back to our "blueprints"

Templates Defined

- So how does all of this relate to templates?
- Well, the notion of abstracting out a function definition into pseudo code such that any type could be substituted is exactly what the concept of (function) templates is all about.
- If only the syntax were a little easier to look at! "sigh"
- Consider the following C++ function template definitions of swap() and sort():

```
template <class someType> void swap(someType *a, int i, int j)
{
    someType temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

```
template <class aType> void insertionSort(aType *a, int size);
template <class aType> void selectionSort(aType *a, int size);
```

What is a Template?

```
template <class aType> void selectionSort(aType *a, int size);
```

- 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.

Calling Function Templates

- A function template is called just like any other function.
- There is an optional syntax for allowing the default "type resolution" to be overridden.
- Consider the following code:

```
class Person {
    public:
        void whoAmI() { cout << "I am a Person!" << endl; }
    }

class Student : public Person {
    public:
        void whoAmI() { cout << "I am a student!" << endl; }
    }

template <class someone> whoAmI(someone *x, Person *y) {
    x->whoAmI();
    y->whoAmI();
}
```

- In addition to calling whoAmI() like a normal function, I can also override the type of the argument being passed as the first parameter, like this:

```
int main()
{
    Student alex;
    whoAmI(sales, &alex);
    // Even though "alex" is a "Student", treat like a "Person"
    whoAmI<Person>(sales, alex);
}
```
Demonstration #1

Calling Function Templates

Demonstration #2

Simplifying INT

Overloading Operators with Templates

We can take advantage of some existing global templates in the Standard Template Library to simplify INT.

You see, operators can be overloaded with templates as well.

The following templates exist in the Standard Template Library:

```cpp
template <class T>
inline bool operator!=(const T& x, const T& y)
{ return !(x == y); }
template <class T>
inline bool operator>(const T& x, const T& y)
{ return (y < x); }
```

Overloading Operators with Templates

If you'll notice, this provides a definition for the operators !=, >, >= and <= in terms of == and <.

This means that for any class you create that can be ordered, all you need to do is overload == and < and you'll get all the comparison operators thanks to these templates.

```cpp
template <class T>
inline bool operator<=(const T& x, const T& y)
{ return !(y < x); }
template <class T>
inline bool operator>=(const T& x, const T& y)
{ return !(x < y); }
```

Specialization

Sometimes having a generic solution applied across all types doesn't work the way you want it to.

Consider the example of a template function used to do generic comparisons:

It works fine for most types, but when we start using C style strings, it breaks down:

```cpp
// A generic template to compare two parameters of same type
template <class aType>
bool isLessThan(aType arg1, aType arg2)
{ return arg1 < arg2; }
```

```cpp
int main()
{ int x = 7, y = 2;
  char *str1="oregano", *str2="basil";
  if (isLessThan(x,y)) // This is OK
    cout << "x less than y" << endl;
  if (isLessThan(str1,str2)) // Is this what we want?
    cout << "str1 less than str2" << endl;
}
```

```cpp
bool isLessThan(char *str1, char *str2)
{ return (strcmp(str1,str2) < 0); }
```

The problem is that our generic comparison routine will be comparing the pointer values for str1 and str2, not the strings they point at.

If only we could make a special case… (can we?)

```cpp
bool isLessThan(char *str1, char *str2)
{ return (strcmp(str1,str2) < 0); }
```
Demonstration #3

Specialization

Lecture 17

Final Thoughts