Introduction to Templates

Where are we in the course? The first third of the course talked about memory, and culminated in the first midterm. The second third talked about object oriented programming, and culminated in the second midterm. (Yes, I know it’s a misnomer to have two “mid”terms, but hey.) We have one more major topic to talk about: templates. This will be the focus of the remaining third of the course.

But don’t forget memory or objects. Everything comes together.

So let’s start talking about templates. First I want to set up a contrast by talking about generic programming in Java. I am not a Java expert, but your book confirms my somewhat week knowledge of Java.

How do you make abstract collections in Java? Using polymorphism and/or generics. Let’s look at the polymorphism approach first.

Let’s say you want to make a vector that collects instances of any type of object. The Java approach is to use the singly-rooted object hierarchy and polymorphism. In other words, you make a vector of “objects”, the root of the hierarchy.

This works because Java doesn’t directly build a vector of objects, it builds a vector of references to objects. Since references are a constant size (even if the object sizes vary), the references stored in the vector are all the same size. You can build a vector or hash table or any class you like that collects other classes, because references are always the same size.

It has some disadvantages, however. Because you are manipulating references to objects, the objects may be spread all over memory. Locality of reference is gone, which may mean less efficiency (the common Java trade-off).

This approach also means that you can’t put primitives in your collection. Since primitives are handled differently than objects – in particular, they aren’t accessed through references – you can’t put them in your collection. To work around this problem, Java added wrapper classes like ‘Integer’, which are objects that contain a single primitive. But this is even less efficient, and its ugly, since the integer you stored and the primitive int in your program aren’t the same data type.

It also leads to lots of down-casting. Every time you take an object out of the vector, you have to cast it back to its original data type (some child of object). This means extra code. It means a loss of efficiency, since run-time type checking is slow (similar to dynamic_cast in C++). And it means that a lot of errors aren’t caught until run-time.

To avoid the last problem, Java has introduced generics. Syntactically, generics were designed to look like C++ templates, in that they enclose a data type in angle brackets. But they don’t do quite the same thing.

A generic in Java (as I understand it) is a directive to the compiler about the type of object being stored in this instance of a collection. In other words, the vector class is written without knowing what class will use it. When the generic is instantiated, it is instantiated with a type. This does two things:
• It avoids the need to down-cast. It knows the type (or at least a parent of the type – there is still polymorphism going on), and therefore it knows the type of instances that you take out of the collection. This is more efficient than run-time type casting.
• It converts data type errors from run-time errors to compile-time errors.

Generics are a good addition to Java. They alleviate two of the downsides of using polymorphism to create collections (down-casting and run-time errors). The underlying mechanism is still polymorphism, however, so the problems of no data localization and differential handling of primitives still remain.

C++ has a more strongly typed approach: templates. Despite the syntactic similarity, templates are not the same thing as Java generics. Generics are still based on polymorphism. Templates are based on meta-programming.

As usual, it is possible to mimic Java style in C++, in this case by using polymorphism to create general containers. For example, I could make a Number class that had multiple extensions, e.g. Complex and Rational. Then I could make a vector class that held references to Numbers, and use it to hold references to both types.

But it’s harder in C++ than Java. First of all, I have to worry about splicing, so I can’t make a vector of numbers, I have to make a vector of references to numbers or pointers to numbers. Note that Java does this too, but it’s automatic so you don’t have to worry about it. In C++, you have to do it all yourself. And if you use references, the lack of locality of reference becomes an issue (as it is in Java).

Also, there isn’t a singly-rooted hierarchy in C++. If you want a truly generic collection (a.k.a. container), you have to build your classes into a hierarchy, including defining classes for Integer, etc. (Which you can do better than in Java because of overloading, but still it would be a lot of work.)

So the more common C++ approach is based on templates. Templates are fundamentally different. They are not a polymorphism based approach. Instead, a template is meta-code: it is a definition for the compiler of how it should define new data structures and classes as needed.

For example, let’s write a simple templated function:
```c++
template <typename TYPE>
TYPE Max(TYPE arg1, TYPE arg2)
{
   if (arg1 > arg2) return arg1;
   else return arg2;
}
```

First, a syntactic note: the template command applies to the ONE C++ expression that follows it. In this case, the expression is the definition of Max.

But note that this is a template definition, not a function definition. No source code is generated directly as a result of this templated expression.

Instead, this is meta-code for the compiler. It says, “If a program uses a function called Max on two arguments of the same type, and you don’t already have a function that matches this specification, here is how you should define it.”
So the template is a fill-in-the-blank structure. If later in the code I write:

```c++
Int a = foo();
Int b = bar();
Int c = Max(a, b);
```

Then the compiler checks if I have a function called Max that takes two integers as arguments. It doesn’t find one, but it does find the template above, so it generates a new function based on the template. That function looks like:

```c++
int Max(int arg1, int arg2)
{
    if (arg1 > arg2) return arg1;
    else return arg2;
}
```

So it the compiler defines and compiles the function Max(int, int) when it sees it used. The function Max(int, int) is now just like any other function.

Note that no polymorphism is being used here. It didn’t find a Max function for integers by casting them up the hierarchy. It defined a new function that operates specifically and exclusively on ints.

This means that the greater-than comparison can be statically dispatched. It knows that this is the int version of ‘>’.

More generally, it has a specific function with fixed data types that it can optimize like any other function. Does it have to step through an array? No problem, it knows how large the elements are. Does it have a ‘> 0’ test on an unsigned data type? If so, it can optimize that away.

This is the big difference: templates produce strongly-typed functions and classes that can be optimized. Java generics produce one function that can be used for all types; C++ templates produce N specific functions, each optimized for a specific type.

Let’s go back to the max template for a bit. What if I write:

```c++
Quagga a = foo();
Quagga b = bar();
Quagga c = Max(a, b);
```

Will this work? Depends if I have overloaded the > operator for Quagga’s. If I have (to compare weights, for example), then c will become the heaviest Quagga. If I haven’t, it’s a compile-time error. Why compile time? Because the compiler generated a version of Max for Quaggas, only to compile it and find it couldn’t find the > operator.

Templates can be applied to classes as well as functions. In fact, this is what you’ve been doing ever since the first assignment: vector is a templated class.

We can write our own version of vector now:

```c++
template<
    typename ELEMENT>

class vector {
public:
    vector(unsigned int initial_size = 0);
    ~vector();
};
```
void push_back(ELEMENT element);
ELEMENT& operator [] (unsigned int index);

private:
    unsigned int size;
    unsigned int capacity;
    ELEMENT* data;
};

Note that the template definition still applies to one C++ expression, but now the expression is class vector { ... };

Note also that ELEMENT appears as an argument (e.g. to push_back), a return type (e.g. operator []), and a data field (e.g. data). Note also that ELEMENT* and ELEMENT& are OK, since you will get a data type when you substitute for ELEMENT. Also, it OK that some methods and fields do not use ELEMENT, for example ~vector() and the size and capacity fields.

We can start to implement our vector class as well.

template<typename ELEMENT>
vector<ELEMENT>::vector(unsigned int initial_size)
: size(initial_size),
    capacity(2 * initial_size),
    data(new ELEMENT[initial_size])
{}

Notice the syntax on the class name (vector<ELEMENT>::). The data type includes the element specification.

template<typename ELEMENT>
vector<ELEMENT>::~vector()
{
    delete [] data;
    data = NULL;
    size = 0;
    capacity = 0;
}

Even though the destructor does not explicitly mention ELEMENT, it has to be declared for type vector<ELEMENT>. Partly, the data type is implicit. The call to delete has to know the data type of the pointer being deleted. The data type is ELEMENT (from the class definition). So the destructor is using the data type. Also, there is no class called vector. Just vector<ELEMENT>.

template<typename ELEMENT>
void vector<ELEMENT>::push_back(ELEMENT element)
{
    if ((size+1) < capacity) {
        data[size] = element;
        size++;
    } else {
        capacity *= 2;
    }
```cpp
ELEMENT* new_data = new ELEMENT[capacity];
for(unsigned int i=0; i < size; i++) {
    new_data[i] = data[i];
}
delete data;
data = new_data;
data[size] = element;
size++;
}

template<typename ELEMENT>
ELEMENT& vector<ELEMENT>::operator [] (unsigned int index)
{
    return data[index];
}
```

Things to note:
- Qualification doesn't look any different
- Types have to match only after substitution
- There is nothing primitive about vector, string, or any other class in the STL. They are just classed implemented in C++ using templates. In fact, STL started as a corporate template library at SGI (then at HP). It has slowly been added to the language as a set of required library files that can be included.

Templates and polymorphism can be combined. I could, for example, go back to my number class with two extensions (rational and complex), and declare a vector of pointer to number. It has to be a pointer or reference, of course, because of slicing.

Rule of thumb: if you want to store a collection of like types, just use a template with no polymorphism. Only add the polymorphism if you need it.

For those of you with some C experience: templates are a much more powerful form of substitution than the relatively weak C macro facility.

A practical note: where (in what file) do you put a template?

I recommend putting the entire template in one .tpf file (declaration & implementation). This file is included in other files. Each file that includes it compiles instances of the template as needed. If two files include it and both instantiate it for the same data type, they will redundantly compile it, but the linker will include only one version in the final executable program.

The disadvantage to this approach is that compilation may take longer for large systems with templates.

This is the approach used by the STL (although they don't use the .tpf extension).

This is only one possible convention for templates (called the Borland Approach, although Gnu also recommends it).

Another convention is called the Cfront Model. In this approach, the template class is in a .h file, and the implementation is in a .cpp file. Programmers then add explicit instantiations of templates as needed at the end of the header file. The .cpp file compiles all the implementations only once.
However, you have to know how you are going to use the templates and put that into the header file. Every time you create a new instantiation you have to go back and edit the template file. Which means you need permissions (think of the STL files). I don’t like this convention.

Finally, there is a new approach that might catch on. In this approach, templates are split into .h and .cpp files (as in the Cfront model), but no explicit instantiations are included. Applications use a #pragma statement to declare instantiations of templates. A utility function called inside the makefile scans your program for the pragmas and automatically creates a file listing all instantiations for each template type. The .cpp file of the template reads this file and declares the appropriate instantiations.

Conceptually this is better, but the tools are still raw: the makefile, utility, and pragmas all have to be just right. When the tools improve, this may become the standard method. For now, use .tpp files.