Programming Notes 19: Templates

As usual, more examples of the notes are in the example program. The example is very important this week! It gives a full overview of templating a data structure.

1. Generic Programming and Data Structure Abstraction

Generic programming is a general topic of discussion in computer science, and it represents the ability to program logic without knowing or caring about the actual type of data you will be using. Generic programming is particularly useful for abstracting data structures to the point that all you have to think about is the logic behind your data's organization, not the data itself. Realistically, this would mean (for example) creating a list data structure that is agnostic of what type of data it stores, whether it be integers, strings, or even lists.

2. Templates and C++

Most languages support most, if not all, of what we've learned so far, although classes (i.e. objects) are specific to object oriented ones. However, we're now going to start covering some C++ specific features, and templates are one of them. Other languages do support generic programming, but C++'s templates are an implementation specific to the language.

So, templates are C++'s way to do generic programming, giving you the ability to make functions and data structures type-agnostic. However, you have to be wary of using them—they may cause more problems than they solve. They are hard to validate in all cases, they can significantly slow compilation time, old C++ compilers don't support them, and error messages from templates are practically incomprehensible. Seriously—template errors can generate a hundred or so lines of error messages from one problem, and give you no straightforward information but the line number of the error.

So, why would you want to use templates? Often, you won't. However, when used in moderation, they can allow you to reuse data structures and methods to significant effect.

3. Template Functions

To declare a templated function, you begin with the word "template," as you might expect. However, you can then add template parameters, the important part. Template parameters, specified in pointy brackets, can be normal parameter data, or the name of a type. To declare a type name parameter, simply define the type as "typename," or "class." For example:

```
template <typename T, int x>
// templated thing
```

Now, in your templated "thing," you can use the template parameters as constants (in the case of regular parameters), or actual types (in the case of type names). This is the generic part: the arbitrary type parameter can be used specify variables, pointers, the function return type, parameters, and anything else. For example, if we wanted to make a templated function that creates an array of a type of a specified length:

```
template <typename T, int n>
T* createArray()
{
    return new T[n];
}
```

Of course, “n” could be passed as a normal parameter, but for the sake of the example it’s a template parameter.

So, since templates are used to abstract the type of data you’ll be working with, when actually using a templated function or object, you must specify what type of data you’re working with. Passing template parameters parallels passing normal parameters: you pass actual values, but this time inside pointy brackets, and before your actual parameters. For example, if we wanted to call the previous example to create an array of 10 integers, we would do:

```
int* array = createArray<int,10>();
```

Here, we pass “int” as the type name “T,” and “10” as the integer template parameter “n.” This will create an array of 10 integers.

4. Automatic Template Parameters

Another useful feature of templates is that if possible, the compiler will automatically detect what types you’re using as template parameters. In these cases, you can leave off the template parameters entirely. For example:

```
template <typename T>
T abs_value(T value)
{
    if(value < 0) return –value;
    return value;
}

abs_value(-10);                    // This will automatically use “int” as “T”
abs_value(5.45);                   // This will automatically use “float” as “T”
```

Templates can be used to create many useful utility functions, for example a to-string function, a from-string function, operator overloads, and more, but the main use is creating reusable data structures in the next section.

5. Template Structs/Classes

Declaring a templated structure or class is much the same as templating a function: you begin with the keyword “template” and the template parameters, then declare the class/struct as usual, except using your template parameters. For example, say we want to create a “vector” structure, which contains N elements of type T. In this case, note that N would not work as a normal parameter, as you cannot create a static array with a non-constant variable. Also, you would want a constructor/destructor, but they are omitted for the example:

```
template <typename T, int N>
struct Vector
{
    T elements[N];

    Vector& operator=(const Vector& src);
    T operator*(const Vector& other);
}
```

Here, we create an array of type “T” with length “N,” and prototype some member functions. Note that “Vector” is still used normally, and “T” can be used as the return type for the * operator.

If you were to implement member functions of a templated struct/class normally, the syntax is very obtuse, as you have to declare the member as a template, and each time you were to write “Vector,” you have to write the type with all template parameters. For example:

```
template <typename T, int N>
Vector<T,N>& Vector<T,N>::operator=(const Vector<T,N>& src)
{
    // …
}
```

Because this is ugly and prone to errors, it’s conventional to simply implement your member functions directly in the declaration of your struct/class, for example:

```
template <typename T, int N>
struct Vector
{
    T elements[N];

    Vector& operator=(const Vector& src)
    {
        // …
    }
    T operator*(const Vector& other)
    {
        // …
    }
}
```

6. Using Templated Structs/Classes

You’re probably going to want to use your fancy new templatized data types. Creating a templated type is very similar to calling a templated function: simply specify your template parameters in pointy brackets. For example,

```
Vector<int,3> vec1;
```

Creates a vector holding 3 integers named “vec1.”

Using your newly created object is no different than anything else you’ve dealt with, member functions, data members, operators, and everything else works the same way.

7. Templated Nodes

As I’ve mentioned several times, templates are often used to create generic data structures. I don’t have much to say here: first, we’ve already created a simple templated data structure, the Vector. Simply templating a struct/class is often enough to create your data structure. However, say you wanted to template a linked list-based structure, like we made last week. Here, you would need to template both the “node,” to hold arbitrary types, and the list to accommodate templatized nodes. SEE THE EXAMPLE PROGRAM