TEMPLATES AND STL

Generics in C++

One of the things that C++ manages to do very well is to provide generic programming to us. The *STL* is one of the best functionalities that have been ever made in the language. `Vectors` , `lists` , `for_each` , and many many more containers are available to use to shorten our code, make it more understandable, and make it independent from types! The C++ template mechanism was also Turing complete (you can compute any computable value with it), which led to template meta programming. With this tutorial we will go over what templates can do, how we can use them (and when) and then we will concentrate on the STL.

Template programming

- Templates
- Compile-time polymorphism
- `typename` , `class` , `typedef`
- Template Meta Programming

Templates

```
template <typename T>
T GetMax (T a, T b)
{
    return (a > b ? a : b);
}
```

```
template<typename T>
void doProcessing(T& w)
{
    if (w.size() > 10 && w != someObject)
    {
        T temp(w);
        temp.normalize();
        temp.swap(w);
    }
}
```

There is a good chance that you will not be dealing with a lot of template code in C++ and that's okay. But it is still a good idea to be able to understand and write templates in C++. In fact, compared to Java or C#, C++ has more flexibility when it comes to writing such code (although this isn`t always good). So what are templates? They are just something that can be substituted for something else during run-time. It allows us to greatly reuse code, abstracts us from dealing with types, but at the same time, it makes executables larger, and sometimes slower.

Okay so let`s look at a template.

In the first piece of code, we have a template function that returns the larger of the two types. These types can be literally anything - they can by `int` , `double` , `char` , `someObject` , etc. As long as the type `T` has an implementation of `>` , this will compile and this will run. In the second example we can see even clearer when we say that some template must support something. We are passing `w` by reference (so maybe it's an object). The parameter uses the `size()` , `normalize()` , and `swap()` functions (okay, it's an object). We also have the operator `!=` in use, and we also have a copy constructor. If we were to just substitute `T& w` with `SpecificObject& w` , we would not worry that `w` supports all of these functionalities, because we have explicitly stated what we are passing. But since we are using a template, we are *expecting* that `T` supports them. This is the meaning of an *implicit* interface. Writing templates means that we are dealing with implicit interfaces in the first place. It's okay not to stress it that much when we are writing generic functions, which are designed to handle build-in types (such as `int` and `double`), but when we are working with user defined types, we have to be careful what the function is actually doing, and does it support the interface.

One thing to consider when dealing with implicit interfaces is that it's not important whether what the function we are using does what it has to do. This means that when we call `w.size()` , this might return some integer indicating the size of `w` , but it might not, it might not ever return a number! As long as the function call returns something that can be used with the `>` operator, this code will compile! Maybe the function is defined in some base class where it is inherited, everything is valid as long as the next operator runs. The same is true for the `!=` operator. It can take a type `X` and a type `Y` and as long as **T** is convertable to **X** and **someObject** to **Y**, the statement will run. So we can say that implicit interfaces are just a set of valid expressions, which impose some constraints. So the constraint of the whole expression doesn't really matter of the types used in it as long as the final yielded result is compatible with `bool` .

Compile-time polymoprhism

Okay we get what an implicit interface is. But next to this there is another very important aspect of templates and that is **compile-time polymorphism**. Sounds scary, it's not. You've most probably have used this and in the world of normal functions this is also known as *function overloading*. There is a debate whether this should also be referred to as polymorphism, but it is what it is. Now where do templates come along in this? Much similar to overloading, when we are declaring multiple templates, we can specify at some point in our code what the template will be (i.e. what type). Since we are telling the compiler the type, during compile time, the type of the template used will be mapped in the object file and will resolve at that time. In contrast to dynamic binding, where during runtime it is decided which function is called, the compiler knows based on the types we use what to substituted for when a function is called.

The STL

- Vectors and strings
- Maps and sets
- Lambda functions

Exercises

Resources

- (Implicit Interface)https://stackoverflow.com/questions/29298212/what-is-an-implicit-interface (https://stackoverflow.com/questions/29298212/what-is-an-implicit-interface)
- (Compile-time polymorphism)https://stackoverflow.com/questions/1881468/what-is-compile-time-polymorphism-and-why-does-it-only-apply-to-functions (https://stackoverflow.com/questions/1881468/what-is-compile-time-polymorphism-and-why-does-it-only-apply-to-functions)
- (Compile-time polymorphism 2)http://advancedcpp.livejournal.com/728.html (http://advancedcpp.livejournal.com/728.html)
- (Mixins)https://michael-afanasiev.github.io/2016/08/03/Combining-Static-and-Dynamic-Polymorphism-with-C++-Template-Mixins.html (https://michael-afanasiev.github.io/2016/08/03/Combining-Static-and-Dynamic-Polymorphism-with-C++-Template-Mixins.html)