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

```
0  template <typename T>
1  T GetMax (T a, T b)
2  {
3      return (a > b ? a : b);
4  }
```

```
0  template<typename T>
1  void doProcessing(T& w)
2  {
3      if (w.size() > 10 && w != someObject)
4      {
5          T temp(w);
6          temp.normalize();
7          temp.swap(w);
8      }
9  }
```

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 Mixin idiom

```
0  template <typename T>
1  class MixinClass : public T
2  {
3      // code
4  };
```

```
00   class MixinBase
01   {
02   public :
03       void f() { std::cout << "HELLO" << std::endl; };
04   };
05
06   template <typename T>
07   class Mixin : public T
08   {
09   public:
10      void f()
11      {
12          std::cout << "call from mixin f method" << std::endl;
13          T::f();
14      }
15   };
16
17  template <typename T>
18  class Mixin2 : public T
19  {
20  public :
21      void g()
22      {
23          std::cout << "call from mixin2 g method" << std::endl;
24          T::f();
25      }
26  };
27
28  int main()
29  {
30      Mixin2<Mixin <MixinBase> > mix;
31      mix.g();
32
33      return 0;
34  }
```

Now that we know about implicit interfaces and the fact that template must support the interface in the method body, we can talk about one of the most useful concepts in C++, which is the **Mixin**.

Mixins are a great way to reuse code and overcome come implementational problems that *composition and delegation* and *inheritance-based-reuse* bring. From a syntactic point of view this is a `mixin`. This is a template class that inherits from the type of the template. Simple enough, but why is this important? Well, mixins are like putting all of the reusable code in the derived class (instead of the conventional way of putting the reusable code in the base class). Basically, mixins are small classes that implement a very specific functionality for another class, through a specified class that provides the specific features the functionality needs.More abstractly speaking, mixins are small fragments of a class that are intended to be composed with other classes. Let's say we are trying to model something, we start writing our code and implement all of our classes and hierarchies normally as we know, at which point we have several classes, orthogonal to each other (`decoupled`). We might have some common interface between those classes and at that point we inherit from that interface. We writing our explicit implementation of what we have inherited. Everything seems fine, but the problem with this design is that we do not have an easy way to combine our modeled classes with each other. This is where mixins come in play. They allow us to create primitive classes that are used as building blocks. We can plug in any class we want into them and get the functionality we desire, while keeping the classes orthogonal to each other.

From the example we can see how a mixin can be used to *"plug-in"* classes together and get some functionality out of them. The code works like this - we create an object that is combined from a base class that just outputs a string and two other mixin classes. At this point we can see how we are usgin multiple inheritance, i.e. inhereting from the template class. We can simply traverse the output of the program and see how using the `g()` method we get the output of the

`Mixin2` class, which, then using the template `T` calls the `f()` method. Since that `T` is of type `Mixin`, we go to that class and execute `f()`. In that method call we also have another `f()`, but this time the `T` of the template is of type `MixinBase`. Sort of a long traversal, but nothing too complicated. This is how we use mixins in the simplest form. Now let's look at a more practical example.

Remember, don't use mixins unless your programs needs them. Although their versatility, they bring complexity to the whole structure.

`template` or `class`

```
0  template <class T> class Thing;
1  template <typename T> class OtherThing;
```

One of the things you might have noticed until now is that, in all of the examples we have seen so far, the keyword `typename` has been used. Now let's talk about the differnece between why and when to use `typename` and when to use `class` when declaring templates and so on. In the example we have, the difference between declaring a template parameter with `class` or `typename` is *nothing*. But there is a logical difference. Some people like the *class* because it's less typing, others like *typename*, because it indicates that the passed parameter doesn't need to be a class type. You can choose to use *typename* when any type is allowed and to use *class* when a user-defined type is passed. So what's the big fuss, when both are considered equal by C++. Well, C++ does find a difference between the two, and sometimes we do have to use only typename, and sometimes we have to use class only. Here are the edge cases.

```
0  template <class T>
1  class Demonstration
2  {
3  public:
4      T::A *aObj; // oops! error!
5      typename T::A *aObg; // correct!
6  };
```

For *typename* - During the olden days of standardization of the language, the C++ people found a problem when trying to make declarative statements in a templated class. That is, when we have a template class, in which we use the template to access another nested template, we would get an error. The problem was that when we are accessing nested templates, C++ doesn't recognize we are doing that and it thinks that we are actually multiplying object `A` by `aObj`. In order to fix this, we just have to tell the compiler that we want to declare something (and not break everything). This is the situation when we use *typename*, to indicate a declerative statement.

```
0  template <template <typename, typename> class Container, typename Type>
1  class MyContainer: public Container<Type, std::allocator<Type>>
2  {
3      //code
4  };
```

For *class* - The other edge case is when we have a *template template parameter* declaration. When we are declaring a template template, we are limited to use the `class` keyword in order to tell the compiler that the template parameter is `Container`. If we are to write `template Container`, then we would get a compiler error. This is the way it is... almost. Form C++ 17 this is no longer a problem since writing either `class` or `template` will yield the same valid result. If you are using C++ 14 or 11, keep in mind for this restriction.

The STL

- Containers
- Vectors, strings, maps, and sets
- Iterators
- Lambda functions

Now that we have a good grasp of what templates are, we can concentrate on one of the best things in C++ - the STL (*The Standard template Library*). The STL is an arrangement of containers that allows us to manipulate our data in easy and intuitive ways. Although there is a whole other book specifically for how to use the STL in an effective manner. We can't cover all of those items here, so we will just look over some of the most useful things (or at least to my perception). Nevertheless, after this section you should feel comfortable to dive deep into the world of STL, it's both fun and it makes C++ much much easier (if done right).

Containers

- Sequence containers
- Associative containers
- Unordered associative containers
- Container adapters

The containers of C++ are the fundamentals of how the STL works. These are collections of the same object, where they hold/own the elements in them. Containers are very flexible due to their *template* implementation. We have three general forms of containers. *Sequence* containers control the order in which elements are stored (`vector` , `array` , `list` , `deque`). The order in which we insert our data is the order we are going to get it. *Associative* containers control the position of the elements (`set` , `map` , `multimap` , `multiset`). In this case the elements are not inserted based to the position they have, but on their value. Finally we have *Unordered* associative containers, which holds the position of the elements through a hash to each element (`unordered_set` , `unordered_map`). There are also the so called *container adapters*, which are not full blown containers, but are used to hold some other container in them (`stack` , `queue` , `priority_queue`). These adapters hold the other container and through the adapter do we access the elements. Of course there are many more types of containers, but these are the basics and should get you going.

To `size()` or to `empty()`

```
0  if (cont.size() == 0) // same lines
1  if (cont.empty())     // of code
```

Now let's talk about some implementations we can use with containers, which are going to make our lives easier. One this is that we will often, when using containers, see if it's empty or not. We have to ways of doing that, wither through `cont.size() == 0` or `cont.empty()` . Which one is the better way? The latter is. Because, when we are calling the `size()` function, we are telling the container to traverse through the whole construct to see what is the actual size. The problem here is that, size cannot be implemented to return in constant time. Since new elements have to change the size of the container, this puts the implementer in a conundrum, how to make size return in constant time, and different container implementations make different choices. To overcome this, just use `empty()` , because this method will always return in constant time.

Prefer range member functions to for loops

`v1.assign(v2.begin() + v2.size() / 2, v2.end());`

```
0  std::vector<Obj> v1,v2;
1
2  // code thingies
3
4  v1.clear();
5  for (auto &v : v2.begin() + v2.size() / 2)
6  {
7      v1.push_back(v);
8  }
```

One of the best things about the containers is that they come with built in awesome member functions that will do a lot of work for us with the added benefit of making the code base clearer and smaller. The problem is that we don't always know that the containers we are using have such awesome functions that come along with them. Specifically, if you like writing for loops for containers, then maybe it's time to look over what that specific data structure can do for you. Let's take this example. We have two vectors `v1` and `v2`. We want to make `v1` be the same as the second half of `v2`, what's the easiest way? Answer : `v1.assign(v2.begin() + v2.size() / 2, v2.end());`. Here we are achieving a few things. First we are making our code much smaller and clearer, combining everything we want into just a few characters. Second, we are not using some form of looping logic we wrote, because look at how we would have to write it if we were to make it ourselves. The best way to overcome such problems is to ask yourself before you try to use a loop - "What do I want to achieve with this loop and can I use something from the containers method set?". This will both decrease the code you write and will increase your knowledge of the STL, because most of these range based members are in the majority of containers.

Delete `new` ed objects in containers

```
0  void awesomeFunction()
1  {
2      std::vector<Obj*> v_obj;
3      for (int i = 0; i < 42; i++)
4      {
5          v_obj.push_back(new Obj); // ticking timebomb
6          // more code...
7      }
8  } // memory leak happens here!
```

```
void awesomeFunction()
{
    // .. same code as before

    for (auto &v : v_obj)
    {
        delete v;
    } // no memory leaks!
}

// or

void manageFlights()
{
    std::vector< std::unique_ptr<Plane> > airport;
    for (int i = 0; i < 10; i++)
    {
        airport.push_back(std::make_unique<Plane>());
        airport.at(i)->fuel = 4000;
        std::cout << airport.at(i)->fuel << std::endl;
    }
} // No memory leaks!
```

More often than not, you will be creating objects on the `heap` regularly and you will be using your containers as usual. So you will, naturally, combine both your new objects in the containers, but there is a small problem here. Containers by themselves do handle their own memory and they do delete themselves once their are of no need, but that doesn't mean that they will handle data on the `heap`. That's why if you have collections that are filled with such objects, you have to manually delete them. Now there is some wiggle room and that is that we can use *smart pointer* to our advantage and not worry about memory allocation and deletion. Now there is some clearing up to do. Before *C++ 11*, we had only one type of smart pointer and that is the `auto_ptr`. Since then `auto_ptr` isn't all that smart and caused a lot of troubles. One of which is the fact that we couldn't use it with any container. It was so bad that the C++ standards committee actually forced a rule where the compiler will not allow the program to have an `auto_ptr` in any container. But after C++ 11, things changed and we have `uniquq_ptr`, which should be your default *goto* when dealing with smart pointers. Let's get back to our example. Now we have to general approaches to fix our code leak. One is to manually go over the container with a `for loop` and delete by hand what we have created. Or the other is to use the magical `unique_ptr` and let that handle everything. Generally, use the smart pointers that are at our disposal.

How to **really** erase things from a container

- dependent on specific container
- erase-remove idiom
- normal remove

```
// removes all elements with the value 5 from a vector
v.erase(std::remove(v.begin(), v.end(), 5), v.end());
// remove all elements with the value 42 from a list
c.remove(42);
// remove all elements with the value 68.9 from a map
m.erase(68.9);
```

Why do we need to know that there are 3 types of containers? Many reasons, but one of them is, because this will help us utilize the removal of items in an efficient manner. Every type of container has it's own way to remove elements and there is not general approach, but here are some guidelines. If you have a `string`, `vector`, or a `deque` (i.e. a sequential container), then use the *erase-remove-idiom*. What is the *erase-remove-idiom*? It's a popular technique for these types of

containers where we call the two methods `erase()` and `remove()` to delete something. We can see from the example how to use it. *Note*: that in order to use `remove()` we must include the `<algorithm>` library in our code, since the sequential containers don't have such a method.

For *lists*, the situation is different. Although the above will work, there is a better way. You should just use the built in `remove` method in the list class. In general here is another *rule as a bonus*: prefer the class specific methods of containers to the same methods in the `algorithm` file. This is much more efficient.

For *associative containers* however things are different. We should never use anything that has the word *remove* as a way to delete elements from the container. This will probably lead to data corruption and in-proper element removal. Also such container don't even have a method called `remove()`. Instead we should use the `erase()` method and it will work just fine.

In general the *erase-remove idiom* should work for most things, but with little experimenting you will get the hang of things.

What about if we have a conditional?

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)
- (Mixins 2)https://stackoverflow.com/questions/18773367/what-are-mixins-as-a-concept (https://stackoverflow.com/questions/18773367/what-are-mixins-as-a-concept)
- (Mixins 3)https://stackoverflow.com/questions/7085265/what-is-c-mixin-style (https://stackoverflow.com/questions/7085265/what-is-c-mixin-style)
- (Mixins 4)https://en.wikibooks.org/wiki/More_C++_Idioms/Curiously_Recurring_Template_Pattern (https://en.wikibooks.org/wiki/More_C++_Idioms/Curiously_Recurring_Template_Pattern)
- (Mixins 5)http://www.thinkbottomup.com.au/site/blog/C%20%20_Mixins_-_Reuse_through_inheritance_is_good (http://www.thinkbottomup.com.au/site/blog/C%20%20_Mixins_-_Reuse_through_inheritance_is_good)
- (Mixins 6)https://yanniss.github.io/practical-fmtd.pdf (https://yanniss.github.io/practical-fmtd.pdf)
- (Typename why do we have it)https://web.archive.org/web/20060619131004/http://blogs.msdn.com/slippman/archive/2004/08/11/212768.aspx (https://web.archive.org/web/20060619131004/http://blogs.msdn.com/slippman/archive/2004/08/11/212768.aspx)
- (Typename and class)https://stackoverflow.com/questions/2023977/difference-of-keywords-typename-and-class-in-templates (https://stackoverflow.com/questions/2023977/difference-of-keywords-typename-and-class-in-templates)
- (Difference between typename and class)https://stackoverflow.com/questions/2023977/difference-of-keywords-

typename-and-class-in-templates (https://stackoverflow.com/questions/2023977/difference-of-keywords-typename-and-class-in-templates)