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` or `class`

Templates

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

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 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 some generic type that can be substituted for any build-in or user defined type as long as the implementations using the type are valid. It allows us to greatly reuse code, plus it abstracts us from dealing with types, but at the same time, it makes executables larger, and sometimes *slower*.

```
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   }
```

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 the `>` operator, this will compile and 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, so 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 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 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 are 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 using 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.

The general rule to remember is - 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. A good rule is 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.

```
template <class T>
class Demonstration
{
public:
    T::A *aObj; // oops! error!
    typename T::A *aObg; // correct!
};
```

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.

```
template <template <typename, typename> class Container, typename Type>
class MyContainer: public Container<Type, std::allocator<Type>>
{
    //code
};
```

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. From 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 from my perspective). 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. Often, when using containers, we will check if they are empty or not. We have two ways of doing that, either 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!
```

```
0   void awesomeFunction()
1   {
2       // .. same code as before
3
4       for (auto &v : v_obj)
5       {
6           delete v;
7       } // no memory leaks!
8   }
9
10  // or
11
12  void manageFlights()
13  {
```

```
14      std::vector< std::unique_ptr<Plane> > airport;
15      for (int i = 0; i < 10; i++)
16      {
17          airport.push_back(std::make_unique<Plane>());
18          airport.at(i)->fuel = 4000;
19          std::cout << airport.at(i)->fuel << std::endl;
20      }
21  } // 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 pointers* 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 `unique_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

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

```
 0  bool bad_val(x); // returns whether or not x is a bad value
 1
 2  // removes bad_val if it returns true
 3  some_list.erase(std::remove_if(some_list.begin(), some_list.end(), bad_val), some_li
 4
 5  // same as previous line but for lists specifically
 6  c.remove_if(bad_val);
 7
 8  // bad way
 9  for (AssocContainer<int>::iterator i = c.begin(); i!= c.end(); ++i)
10  {
11      if (bad_val(*i)) c.erase(i);
12  }
13
14  // good way
15  for (AssocContainer<int>::iterator i = c.begin(); i!= c.end();)
16  {
17      if (bad_val(*i)) c.erase(i++);
18      else ++i;
19  }
```

The previous example was if we wanted to remove all elements with a specific value. But if we are to use a conditional statement, under which we want to remove something, we can go about in a similar way. Again we will see how to remove items in the different types of containers.

For sequential containers (`vector`, `string`, `deque`, and `list`) is very similar to the previous method. If we have a *list* specifically, we can use the built in method `remove_if()`.

This is pretty familiar, but there is a catch when we reach the associative containers. There is not

method to do what we want to do, which means that we have to write our own loop. That's not a problem, because the logic is easy to implement, but we might fall into a trap. Looking at the two examples, we can see almost similar code. The first snippet seems pretty intuitive, and it is, but it will yield **undefined** behavior. Because we are erasing `i` at the time we find the `bad_val`, that erasure will make all iterators that point to the value *invalid*. This means that once that piece of code runs, the returned `i` will be invalid, making the code *undefined* since we are then using `i` again to increment. To overcome this we have to keep in mind the position of `i` in order to never *invalidate* it. Thus, we remove the incrementation from the `for` statement, and handle it in the `if` statement. Once we reach the bad value, just before we erase the value, we pass the old *unincremented* value of `i` and return the new *incremented* one. We do this with the postfix notation. If we don't have a bad value, then we just increment normally `i`.

From my perspective, this is one of those moments where C++ isn't as consistent as we want, because we are doing the same thing for different containers with different code. It's not that bad and once we start writing the code we will get the hang of it, but we still have to remember what are the optimal way to do this job.

Vectors, strings, maps, and sets

Until now, we just talked about the general idea about containers and how to work with them. Every container has a specific job it does best, and most containers have some general use. These four are the most popular. We can't cover all of the containers in a single swoop, but having an idea of the most popular and useful ones will give you courage to experiment and dig deeper into the STL. Now let's get this out of the way, don't use `array` or `char*`. Using those two things will bring you a lot of headaces and will just leave you with resentment towards C++. When I started learning C++ all I did was use arrays of different types and it was just a paint - you have no dynamic memory allocation, you had to be careful when you are indexing and reading/writing elements, the syntax is annoying, etc. Instead you should adopt the use of `vector` and `string` anywhere you have a sequence of elements or a sequence of characters. Underneath, both of these structures do use arrays and char pointers, but we don't want to deal with that. I encourage everyone to just look over the specifications of `vector` and `string` and to get familiar with the methods. Here we will look over some ways to utilize these STLs as best as we can.

In order to really drive the point home, here is what happens, when we are using normal arrays in our code. We have to remember to delete everything in the array that we have put, then we must use the correct form of delete (`delete` or `delete []`), then we must make sure that we delete the elements only once. Even if we are not making an array of a type `T` (let's say we are using `int`), we still encounter the problem of having to know in advance how much space we need, let alone the situation when we have to iterate over the elements to do some trivial thing. So anytime you decide to write `new T[..]`, think about if you can't use `vector` instead.

Reserve space for optimization

```
0  vector.size()
1  vector.capacity()
2  vector.resize(size_t n)
3  vector.reserve(size_t n);
```

So lets' talk about how we can populate collections. We can further increase the usefulness of the STL in order to minimize unnecessary operations and save us some trouble. First of which is this - when we are declaring vector object, we have several way to populate the vector - through *initialization lists*, through `push_back()`, and so on. Although vector can handle it's own dynamic allocation, there is still a way to optimize things. Because vector calls the underlying `realloc()` function, found int *C*, it goes through a 4 step process - *1.* Allocate new memory that is a multiple of the previous size of the container. *2.* Then copy all of the previous elements of the previous memory to the new memory. *3.* Destroy the objects of the old memory. *4.* Deallocate the old memory. If we are increasing the size of our vector in excess, we will call all of these steps, which will naturally lead to a slower running program. Minimizing this behavior will be a great thing. Another striking thing is that *every* time such an operation is executed, **every** pointer, reference, and iterator will be invalidated. This is an even better reason for us to make sure that we optimize these processes.

Now before we do that, let's just go over the function that `vector` supports about memory. `size()` just tells us how may elements there are currently in the container. `capacity()` tells us how many elements can be inserted with the already allocated memory. `resize(size_t n)` just resizes the number of elements the container has. If the new number is less than the previous size, then the last elements are deleted and removed, if the new number is greater than the previous size, the remaining empty objects are called with a default initialization. If the new number is larger than the current capacity, there will be a reallocation operation in order to make space for the new elements, before they are inserted. `reserve(size_t n)` forces the container to change to `n` given than `n` is no less than the current size of the container. This typically forces a reallocation, since we want to increase the seize, but if we are to shrink the size, the call is almost always ignored.

```
0  // normal code
1  std::vector<int> v;
2  for (i = 0; i < 1000; i++) v.push_back(i);
3
4  // better code
5  std::vector<int> v;
6  v.reserve(1000);
7  for (i = 0; i < 1000; i++) v.push_back(i);
```

So instead of doing meaningless re-allocations, object constructions, destructions, and so on, we can use the power of `reserve` to tell C++ in advance what we will need. So if we are to push, let's say 1000 integers into a vector, we might write a simple code that does that. The first piece of code will do around *2* to *10* re-allocations, since each time a re-allocation is needed, most compilers increase the size of the container by a factor of the previous size. But if we are to use the reserve method we would do *0* re-allocations. Of course this can be translated if we are using objects which would cause even

more time saving.

***Shrink-to-fit* or how to decrease the size of a vector that is too big**

```
0  // vectors
1  std::vector<Object>(myVectOgObjects).swap(myVectOgObjects);
2
3  // strings
4  std::string s;
5  string().swap(s);
```

Here is another cool little trick. Imagine that you have a very large vector that holds a lot of data initially. But after you have processes some of data you sort (or partially sort) your vector and decide that you want the first few elements. You delete the unnecessary elements from the vector by calling a ranged base `erase()` method (as you should), but although the size of the vector did shrink, it's capacity is still big (unchanged). So you can use the simple, but non-intuitive trick of *swap-to-shrink-to-fit*. It's strange that we have to explicitly call `vector<T>` in order to swap and shrink, but what is happening is that in that place, there is a hidden temporary vector that is the exact copy of the one we want to shrink (created through the vector copy constructor), and after the method call, we get the exact size and capacity we want for our original vector.

Iterators

```
 0  std::ifstream inputFile("interestingData.txt");
 1  std::string fileData((std::istream_iterator<char>(inputFile)),istream_iterator<char>
 2
 3  // ok code
 4  std::ifstream inputFile("interestingData.txt");
 5  inputFile.unset(ios::skipws);
 6  std::string fileData((std::istream_iterator<char>(inputFile)), istream_iterator<char
 7
 8  // better code
 9  std::ifstream inputFile("interestingData.txt");
10  std::string fileData((std::istreambuf_iterator<char>(inputFile)),istreambuf_iterator
```

Apart from the normal data structures that the STL has to offer, we have also one more interesting topic to look at - iterators. Basically, iterators are some object that points to an element from a container. There are four types of iterators (`iterator`, `reverse_iterator`, `const_iterator`, `const_reverse_iterator`). Iterators are written in the style of `T*`, where for each type of iterator, there is a similar way of implementing it. The suggestion here is simple, prefer to use only the simple iterator that C++ gives us. There are several reasons for this. First, the option for us to insert or erase an objects into a container in a specific position accepts only iterators of type `iterator` (not any of the other kinds). Also, although we can transform one type of iterator into anther, this transformation is not always possible. At this point we might not even bother and stick to our normal iterators.

Aside from the general use of iterators I want to mention something that is mostly looked over and should be used more often. It has made my life much easier when it comes to command line user input and that is the `istreambuf_iterator`. Until now we are used to getting input from the console by using the `std::cin <<` operator. This is fine until we realize that the operator omits white spaces, and the moment we need to use a blank space, we hit a wall. One way to overcome this is to disable the option to skip white spaces through the `ios` namespace. But since that approach is kind if slow (maybe even up to 40%) slower. That's why we can just use the `istreambuf_iterator` which is much more intuitive to use.

Lambda expressions

```
0  []() { . . . } // empty lambda function
1  // or
2  [] () mutable -> T { . . . } // T is a return type
3
4  [] // capture list
5  () // argument list
6  {} // function body
7
8  // capture list options
9  []   // empty list, no access from locals
10 [x]  // access to locals by value
11 [&x] // access to locals by ref
12 [&]  // access to any local var by the reference
13 [=]  // access to any local var by the value
14 [=, &y] // mix them together
```

Since C++ 11, we have this neat thing called *lambda functions* or *lambda expressions*. If you come from a background of *Java* or *C#*, these might sound familiar to you. So what are they and what kind of problems do they help us to solve? In some simple terms, *lambda functions* are small snippets of code, also typically called an anonymous function objects, that are never going to be reused and are not worth bothering to name and define. This means that if we want to make some operation over a container, instead of creating a separate `namespace` or `struct` or a `class` where we will have this lonely *one-time-use* function, we can just use a lambda function and be done with it. They can be used instead of a named class with an `operator()`. Here is the basic syntax of a lambda function. We have several parts of a lambda function - the *capture list*, the *argument list*, an optional *mutable* specifier, an optional *return type* `->`, and the *function body*. The argument list and the function body work like any other function so we don't need to explain them, but the capture list and mutable specifier are pretty interesting. The capture defines *what* from the outside of the lambda function can be used inside it and *how* (either to be passed by reference or by value). There are several ways we can use the capture list - we can specify an empty list `[]`, which tell us that no local names from the surrounding context can be used inside the function body. For such a list, data is taken from the passed arguments or from non-local variables. We can use `[&]`, which tells us that we can use all local variables *by-reference*. We can also have an implicit capture by value with `[=]`, where all local variables can be used as copies

(*by-value*). We can have an explicit capture list like `[capture-list]`, where we specify the names of the local variables we want to use. Here if we use the `&` sign we tell C++ that the captured variables are to be used with a *reference*. This can be expanded to be like `[&, capture-list]`, where we specify that all local variables are to be used with a *reference*, except the specified ones in the list, which are to be used by value. With the same logic we can have `[=, capture-list]`, and you can guess what this means (all local variables are to be used by value, except the ones in the capture list, which are to be used with a reference). Of course you can mix all of these together with a *comma separator*.

Now what does the `mutable` keyword do in this situation? Well this allows us to modify the whole lambda expression in the function body, i.e. change the lambda's copies of variables that are captured by value. Usually we don't want to do this, so by default this option is turned **off**. This comes in handy not all that often, but what it allows us to do, is to change variables in the scope of the lambda body when the variables are passed by value. Because underneath, function objects with `operator()()` are implemented with a `const` member function, this is why the default behavior of the lambda function is to also use variables as a constant. If we wanted to change a variable (that is passed by value) and not use the `mutable` keyword, we would get a compilation error.

So what about return types? Well, like any other function, we can specify the lambda to have either an implicit return type or an explicit one. Sometimes the function can deduce what we want on its own. For instance, if we lack the `return` keyword, the function will return `void`. If we have, as an example, `return x+y;`, the lambda will be of that return type. But sometimes, we will have a complex situation, where C++ cannot deduce what we are going to return, that's why we can use the optional form `->` to tell it what it should expect. Let's say we have this sitation - `auto f = [b]() ->int { if (b) return 1: else return 2; };` . In this case we have to explicitly tell C++ what to expect, since the body is too complicated.

Exercises

- Reddit - Daily programmer
- 189 Programming questions
- My coding challenges

Instead of just listing a ton of links to challenges and exercises, here are a few links that will provide much more. One is a sub-reddit, specifically dedicated to solving coding challenges. Most of them are doable in just a few hours. My advice here is to do the easy ones, because they will *"force"* in one way or another to experiment with the *STL* or with object orientation. Although the harder challenges are very fun, they are more about concept, where as the easier ones a just to use some basic algorithm, where the STL is the perfect place for such practice. The pdf is a famous book filled with coding tasks that you might see in an interview, think of them as small tasks about specific topics in each chapter. Whether you have chosen the pdf or the reddit board, just force yourself to experiment with new concepts and new STL features. The best way to learn a language is to practice, practice, practice. Finally I have also shared my personal *github* repository where I occasionally solve such tasks. By no means are they well solved or optimal, but at least you can see somebody else's code and can learn from my mistakes.

Resources

Templates

- Implicit Interface
- Compile-time polymorphism
- Compile-time polymorphism 2
- Mixins
- Mixins 2
- Mixins 3
- Mixins 4
- Mixins 5
- Mixins 6
- Typename why do we have it
- Typename and class
- Difference between typename and class

STL

- Containers
- Containers 2
- Containers 3
- Smart pointers
- Unique pointer
- How to use unique pointers in containers
- Erase-remove idiom
- Erase-remove idiom 2
- Erase elements from a vector
- Iterators
- Why use iterators

Lambdas

- Mutable
- Anonymous functions
- Lambdas
- Lambdas 2
- Lambdas 3
- Functors
- Function Objects
- What is a lambda expression
- Lambda with captures