RESOURCE MANAGEMENT AND GOOD PRACTICES

How to manage memory

When we first learn C++ for the first time we have way too many things going on so we are usually told not to worry about memory management (or we are not even aware that such a thing exists in the language). This is normal, since the language itself is hard enough to learn so we push thins topic for another time. But that time usually comes way too late and we don't have the discipline to think in therms of memory management. Luckily for us, we are writing modern C++, which means that there are a lot of tools deisgn to help us out with this task and to make it much more easy, than it was back in '89.

In C++ we have several resources we can manage, where we are not only limited to some dynamically allocated object. We care for files, database connections, mutexes, GUIs, sockets, etc. This means that having memory leaks is just one of our problems, but being unable to connect to a databse, or corrupt the contents of a file, because we are not careful enough is a whole other story. Here we will try to look at how we can use the STL to our advantage and manage resources manually.

RAII

```
0  // without RAII
1  RawResourceHandle* handle=createNewResource();
2  handle->performInvalidOperation();  // this will throw an excpetion
3  // some code
4  deleteResource(handle); // resource leek never cleaned!
5
6  // with RAII
7  class SomeResource {
8  public:
9     SomeResource(RawResourceHandle* rawRes_) : rawRes(rawRes_) {};
10    ~SomeResource() {delete rawRes; }
11 private:
12    RawResourceHandle* rawRes;
13 };
14
15 SomeResource handle(createNewResource());
16 handle->performInvalidOperation();
```

In C++ we have this magical idiom called **RAII**, or *Resource Acquisition Is Initialization*. This binds the life cycle of a resource that must be acquired before use, to the lifetime of an object. Again, we are not just talking about pieces of memory, but sockets, database connections, file descriptors, mutexes, things that are considered as a resources, are bounded to an object. Then, functions are able to access that resource (through the object), use it, and when it is no longer needed, the object manages the state

of the resource, and closes it appropriately. You might also see this idiom as *Scope-Bound Resource Management*. The important thing here is that even when we get an exception during some constructor, the already created base objects are also released in reverse order of initialization. We are both managing resource leaks and exception safety!! C++ provides us with some ready-made solutions to this principle. `vector`, `string`, and `thread` manage their resources on their own so we don't have to worry about them, but we also have `unique_ptr` and `shared_ptr` where we can manage resources and not worry about their releasing and exceptions.

When to use `new` and `delete`

```
 0  int * ipt = new int;
 1
 2  std::cout << "ipt = " << ipt << std::endl;
 3  std::cout << "*ipt = " << *ipt << std::endl;
 4  *ipt = 1000;
 5  std::cout << "*ipt = " << *ipt << std::endl;
 6
 7  delete ipt;
 8
 9  std::cout << "ipt = " << ipt << std::endl;
10  std::cout << "*ipt = " << *ipt << std::endl
```

When do we actually have to use `new` and `delete`? It's so tempting to just say *never* and move on to the rest of the topics, but this is not exactly true. It's *almost* true. First let's look at what `new` and `delete` do. `new` is the keyword we use in C++ that allocates a piece of memory in the *local store* and has a pointer to that piece of anonymous data. `delete` on the other hand does the opposite and removes the contents of a cell of memory. So let's look at an example. Here we just allocate an integer to the free store and then we read the pointer address and the address contents. The question is what would happen after the `delete` call? Would we have to same contents or would we get the same number again? The answer is - we would get **undefined** behavior. After you call delete, you can still use the pointer for other things, but never ever try to see what it points to, since we will go down the valley of *undefined* behavior.

So using `new` and `delete` can lead to peril if we are not careful, so what are our other options?

A pass by over garbage collection

- Weak references
- Reference counting
- Rule of five

Weak references

Let's just go over some interesting and important notes about garbage collection and memory management before we continue with the main topics. Why doesn't C++ have a garbage collector like `Java` or `C#` or some other languages? Well it's not impossible and there are some implementations of a C++ garbage collector, but it's not something that is used widely in the C++ sphere. There are a lot of reasons why this is so, time constraints, agreements between the C++ implementation group, implementation problems in general, etc. The thing with garbage collection is that it's a pretty heavy process and it isn't as simple as pushing everything into smart pointers and calling it a day. C++ is above all efficient, it gives us a lot of room to make whatever we like. The garbage collection we have is the explicit one we can achieve through smart pointers, but there isn't a thing like the `Java` or `C#` collector (an implicit GC) (As a side note, I have included a link to a *C++ GC* if you still want to try it out). So the question here is - how do smart pointers work in C++. Everybody is talking about them, they do have great functionalities, but how are they actually helpful?

In order to understand these concept we need to cover several topics. Let's start with *weak references*. In general, abstracting ourselves from C++, a *weak reference* is a pointer that does not protect the content that pointer points to from a garbage collector. If we have a chain of references to an object and even if one of those references are weak, the whole chain is considered to be weak and can be collected at any moment. Now there are several layers of weak references and we can find them in implementations in `Java`, `C#`, `python`, etc. In general, there are two types of GC - *tracing* and *reference counting*. We are going to look only *reference counting* since it's relevant to us.

Reference counting

Now **reference counting** is a technique where we count the number of references, handlers, and pointers to a certain object, memory block, file, and in general - any resource. Once the number of references reaches *0*, that resource is collected and destroyed. An interesting aspect of this is that when we destroy an object that is no longer referenced by anything, if that object holds a references to another object, after the destruction of the de-referenced object, the other referenced object have their reference count decremented. So there is also a technique where we can add the no longer referenced objects to a list, which we regularly (and by we I mean the GC) visit and destroy some objects from it. All of these processes require constant updating. There are several advantages to this technique, although it being simple - for one, it's very responsive as it doesn't waste any time in de-allocating space for us, thus making it adequate for systems with limited memory. Also the performance of this doesn't go down as the memory is freed. The problem to all of this is that this algorithm takes up memory spaces for the counts, it is not the most efficient method, that is why it's so simple, and it also can't prevent *reference cycles*. A *reference cycle* is just an object that refers to itself *directly* or *indirectly*. This means that their reference count is going to be always above zero. Here *weak references* come in handy since they can break such a chain.

(In this whole segment I managed to re-use the word "reference" 20 times…geez, talk about DRY)

Rule of five

```
0  #include <utility>
1
2  class resource
3  {
4    int x = 0;
5  };
6
7  class foo
8  {
9    public:
10     foo() : p{new resource{}} { } // default ctor
11     foo(const foo& other) : p{new resource{*(other.p)}} { } // copy ctor
12     foo(foo&& other) : p{other.p} // move ctor
13     {
14       other.p = nullptr;
15     }
16
17     foo& operator=(const foo& other) // assignment operator
18     {
19       if (&other != this) {
20         delete p;
21         p = nullptr;
22         p = new resource{*(other.p)};
23       }
24       return *this;
25     }
26
27     foo& operator=(foo&& other) // move assignment operator
28     {
29       if (&other != this) {
30         delete p;
31         p = other.p;
32         other.p = nullptr;
33       }
34       return *this;
35     }
36
37     ~foo() // destructor
38     {
39       delete p;
40     }
41
```

```
42      private:
43      resource* p;
44  };
```

And finally let's talk about the *rule of five*. These are just a set of rules for building *exception safe* and *resource managing* code in C++. The first three rules can be summed up like this. If a class needs to have a user-defined destructor, or copy constructor, or copy assignment operator, then it most probably needs all three. Because we are handling resources in our classes, we will be working with references and pointers, which means that the default implementations of the constructors and destructor will not suffice for us, and we will need to implement them on our own (tutorial 1 covers briefly the topic of writing such custom implementations). The last two rules are also related to this. If we have these custom implementations in our code base, we must also implement a **move** constructor and a **move assignment operator**. Now we must acknowledge what a *move* constructor and *move assignment operator* is. Well basically, they are two new additions we have since *C++ 11*. They allow us to perform move semantics in C++ in a more efficient way (there are links to references for *move* s for those who are interested).

Using the STL to our advantage

```
0  #include <memory> // holds smart pointers
1
2  std::unique_ptr<Object> obj; // holds a pointer to var obj
```

In the previous section I mentioned that we can use smart pointers to our advantage to handle heap memory and resources. Before we had to write out own memory management classes that worked likes smart pointers, when C++ 11 first came out, we had `auto_ptr`, which tried to do that. But it was somewhat impaired to several functionalities like copying and assigning. Since then, `auto_ptr` is deprecated and we use either `unique_ptr` or `share_ptr`. Now I want to get this out of the way, when working with smart pointers, *unique* is the standard and *shared* is the exception.

So how does a `unique_ptr` look like? Well, it's basically using any other STL container, where we define what we want inside of it and then we declare our variable.

General good practices

Exercises

Resources

Memory Management

- RAII

- RAII 2
- Wiki RAII
- Why don’t we have a GC in C++
- Why no GC in C++
- Garbage collector for C++
- Reference counting
- Weak references
- Rule of 3
- Rule of 5
- Rule of 3/5/0
- Move constructor and assignment operator
- C++ 11 move
- Move constructors
- Circular references
- What is reference counting