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.

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 $_{\text{new}}$ and $_{\text{delete}}$? 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 $_{\text{new}}$ and $_{\text{delete}}$ do. $_{\text{new}}$ 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. $_{\text{delete}}$ 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 $_{\text{delete}}$ 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?

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 };
```

```
SomeResource handle(createNewResource());
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 function 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.

Using the STL to our advantage

How to store objects in smart pointers

General good practices

Exercises

Resources

Memory Management

- RAII
- RAII 2
- Wiki RAII