RESOURCE MANAGEMENT AND GOOD PRACTICES

How to manage memory

When we 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 this topic for another time. But that time usually comes way too late and we don't have the discipline to think in terms of memory management. Luckily for us, we are writing modern C++, which means that there are a lot of tools designed to help us out with this task and to make it easier, 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 database, or corrupt the contents of a file, because we are not careful enough, is a whole other story. So first we are going to look at several principles about memory management, how we can allocate memory manually, and then we will move to the STL, where we will take advantage of managing resources automatically.

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  // what happens now?
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. But 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 *de-allocates* the contents to some space 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? Is it possible that we will have 0 as a value or something random? The answer is - we would get **undefined** behavior. After you call delete, that memory is de-allocated and we are leaving it to the computer to decide when and how to manage it on its own. The value might remain, or it might be chnaged immediate, we can *never* know. 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. The problem with this is that there is nothing stopping us from calling `delete` multiple times. We also have to be careful when we are working with *array* since they need some *extra* syntax for deletion. On top of that we must never forget to do this operation.

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 };
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. All of these things 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 (or some operator), the already created base objects are also released in *reverse order* of initialization (going through the inheritance hierarchy). 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 when exceptions occur.

Using the STL to our advantage

- `unique_ptr`

- shared_ptr
- weak_ptr

unique_ptr

```
0  void foo ()
1  {
2      unique_ptr<Thing> p(new Thing()); // p owns the Thing
3      p->do_something(); // tell the thing to do something
4      defrangulate(); // might throw an exception
5  } // p gets destroyed; destructor deletes the Thing
```

```
0  unique_ptr<Thing> p1 (new Thing); // p1 owns the Thing
1  unique_ptr<Thing> p2(p1); // error - copy construction is not allowed.
2  unique_ptr<Thing> p3; // an empty unique_ptr;
3  p3 = p1; // error, copy assignment is not allowed
```

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 instead we have 3 other types of smart pointers - `unique_ptr`, `weak_ptr`, and `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. Since both of them have different implementations, usually we are in need of having a pointer that is unique to a piece of memory, seldom will we need to distribute our resources over many pointers. That's why it's better to prefer `unique_ptr` to `shared_ptr`.

Because these two smart pointers are the most common ones, we can take a look at how they work.

So how does a `unique_ptr` look like? Well, it's basically using any other STL container, where we define a `T` inside of it and then we declare our variable. There are several big advantages to using this pointer. First, it has *zero* overhead. Because it has the `T*` implementation and being able to take in anything, it means that it points to anything and the moment that pointer goes out of scope, the pointer and the pointed data gets destroyed. `unique_ptr` allows us to manage resources without worrying about how they get released and how they get deleted when the time comes. The syntax is the same as using a raw pointer, so we don't have to worry about that. Also, the word *"unique"* isn't arbitrary. This pointer can *"own"* only one object, which means that we can avoid collisions, where multiple pointers can access one piece of data and leave it dangling afterwards. This means that we *cannot copy* nor *assign* one unique pointer to another (things like copy constructors and assignment operators are disallowed for this type).

shared_ptr

```
0  void foo()
1  {
```

```
2       // the new is in the shared_ptr constructor expression:
3       shared_ptr<Thing> p1(new Thing);
4
5       shared_ptr<Thing> p2 = p1; // p1 and p2 now share ownership of the Thing
6
7       shared_ptr<Thing> p3(new Thing); // another Thing
8
9       p1 = find_some_thing(); // p1 may no longer point to first Thing
10
11      do_something_with(p2);
12
13      p3->defrangulate(); // call a member function like built-in pointer
14
15      cout << *p2 << endl; // dereference like built-in pointer
16
17      // reset with a member function or assignment to nullptr:
18      p1.reset(); // decrement count, delete if last
19      p2 = nullptr; // convert nullptr to an empty shared_ptr, and decrement count;
20  }
```

Now that we know what so unique about `unique_ptr`, we can talk about *shared pointers*. The syntax and basic functionality of a `shared_ptr` are the same as any other smart pointer, but there are some implementation differences. For one, we can have multiple pointers that can share the same object on the free store (hence the name *shared*). Using the powers of reference counting, depending on how many references there are to the specific object, C++ can determine when to delete that memory space. This is how it works - once a pointer get out of scope or gets assigned to a `nullptr`, the total reference count gets decremented. If the number reaches *zero*, the object gets deleted. If we are to get into the situation of a *reference cycle*, the smart `shared_ptr` has a companion - the so called `weak_ptr`. These *weak pointers* are used only as observers to the actual object and *do not* affect its lifetime. Because of this, the shared state between the objects is known to C++, the reference counts don't get confused and we get proper object deletion when the count is at 0. But it does mean that when we are using *shared pointers*, we must keep an eye on states where if we have a *reference cycle*, at least one of the pointers should be of `weak_ptr` type, so that we can release the references and prevent memory leaks.

From the above code we can trace how the shared pointer managed and shares the resource it points to. We can decrement the counter manually or through `nullptr` assignment.

A pass by over garbage collection

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 do 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 we work with the explicit garbage collection, i.e. *how do smart pointers work* in C++. Everybody is talking about them, they do have great functionalities, but how are they actually helpful?

- Weak references
- Reference counting
- Rule of five

Weak references

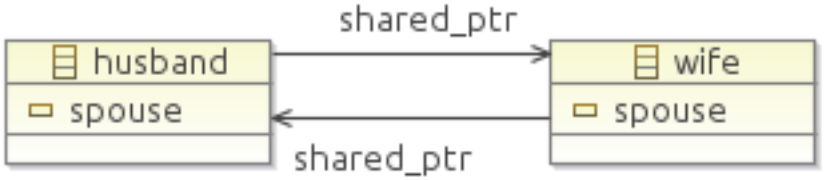

Figure 1: ptr1

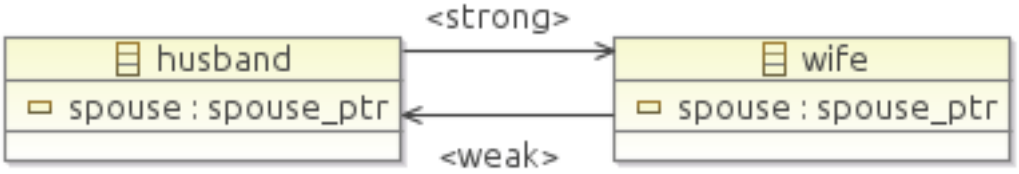

Figure 2: ptr2

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* ones. We are going to look only *reference counting* since it's relevant to us.

Reference counting

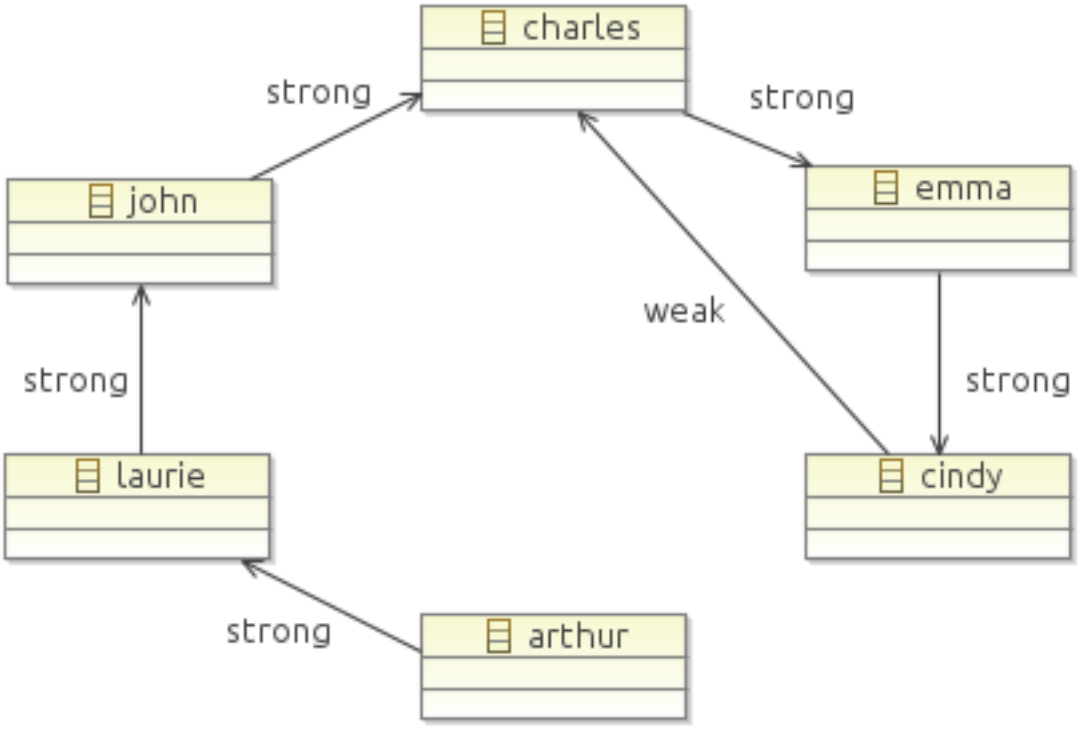

Figure 3: ptr4

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 *subsequently referenced* object has its reference count decremented. So there is also another 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 reference counting, 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 *properly resource managed* 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.

General good practices

Use `const` whenever possible

```
0  char greeting[] = "Hello"; // basic C string
1
2  char *p = greeting; // no const pointer
3                      // no const data
4
5  const char *p = greeting; // no const pointer
6                            // const data
7
8  char * const p = greeting; // const pointer
9                              // no const data
10
11 const char * const p = greeting; // const pointer
12                                  // const data
```

We have already talked about the advantages of *pass-by-reference-to-const* before and we are aware of its benefits. But we can generalize this rule to integrate `const` in more than just function calls. We know what `const` does - it tells the compiler that a specific implementation *cannot* change over the course of the program. Such reinforcement is not only about telling the compiler that, but it also help us follow good semantics in our code. Such constraints also help us to minimize bugs, which might accumulate if we are not careful how we are changing the state of our objects. Outside of classes, `const` can be used to declare global and namespace variables. In classes we can create both *static* and *non-static* constant data members. The problem with this keyword is that it has confusing syntax when it comes to pointers. Because we can declare a pointer to be `const`, we can declare the data the pointer points to be `const`, or make both the pointer and the pointed contents constant. So when we put `const`

before the asterisk in a definition, we are telling the compiler that the data contents are constant, but the pointer can change its "pointing". If we put the keyword *after* the asterisk, the pointer itself will point to only one place and cannot be changed, but the data can be (so we can change the data through another pointer and the constant pointer will still point to the same memory location). Finally we can make them both constants, which just halts a specific part of memory that cannot be changed. *Tip:* it might be helpful, when reading pointer declarations, to start from right to left, since when you say it out loud it would sound natural.

```
0  class TextBlock
1  {
2  public:
3      const char& operator[](std::size_t position) const
4      { return text[position]; } // operator[] for const objects
5
6      char& operator[](std::size_t position)
7      { return text[position]; } // operator[] for non-const objects
8
9  private:
10     std::string text;
11 };
12
13 TextBlock tb("Hello");
14 std::cout << tb[0]; // calls non-const TextBlock::operator[]
15
16 const TextBlock ctb("World");
17 std::cout << ctb[0]; // calls const TextBlock::operator[]
```

Apart from making data members constant, we can also look at constant member functions. The two benefits we get is that we are creating an interface that is telling us what should be what, thus making it easier to understand. The second part is that they will allows us to work with constant objects, thus making our work, by passing by objects by reference to `const` much, easier. There is another thing about constant member functions, and that is that they can differ only on their constness, allowing us overload them based only on that.

Now we should take some time to talk about the two different types of constness we can have in C++. There is *bitwise (logical) constness* and *logical constness*. To be bitwise const means that, inside a constant member function, we are not allowed to change **any** of the class data members. Basically this corresponds to the C++ understanding what `const` is. The compiler just searches in the function body for any assignment operators that might change some *non-static* state in the object. But bitwise constness doesn't always do what it might seem to do. If we have a pointer that points to something and the function is `const`, the data that the pointer points to can still be changed. This means that we can write our functions `const` all we want, but that doesn't guarantee that **all** of the data will remain unchanged. This leads us to the notion of *logical constness*. This means that const member functions can change the values of data members as long as the change is *unseen* from the user perspective. In general it's a good idea to prefer logical over bitwise constness.

```
0  class CTextBlock
```

```
{
public:
    std::size_t length() const;

private:
    char *pText;
    std::size_t textLength;
    bool lengthIsValid;
};

std::size_t CTextBlock::length() const
{
    if (!lengthIsValid)
    {
        // error!
        textLength = std::strlen(pText);
        lengthIsValid = true;
    }
    return lengthIsValid;
}
```

```
class CTextBlock
{
public:
    std::size_t length() const;

private:
    char *pText;
    // mutable allows us to change the values
    // even in const functions
    mutable std::size_t textLength;
    mutable bool lengthIsValid;
};

std::size_t CTextBlock::length() const
{
    if (!lengthIsValid)
    {
        // valid calls
        textLength = std::strlen(pText);
        lengthIsValid = true;
    }
    return lengthIsValid;
}
```

Let's take this example - we want to cache the length of the string we have, which means that inside of

the `length()` function we need to make an assignment, but at the same time the function should be constant (as per good conventions). In this situation, we can see that if we want to save some internal state of the object, we must violate the bitwise constness of the function, that's why we can overcome this by applying the logical constness, and use the `mutable` keyword. It allows us to change the state of a data member even in situations where we have a `const` function. This is what logical constness looks like, because the user has on idea we are changing the internal state of the object, but at the same time, we are following the logic of what we want to do, that is - to save some cached state.

Postpone variable definitions

```
0  std::string encryptPassword(const std::string& password)
1  {
2      string encrypted;
3
4      if (password.length() < MinimumPasswordLength)
5      {
6          throw logic_error("Password is too short");
7      }
8
9      // work
10     return encrypted;
11 }
```

```
0  std::string encryptPassword(const std::string& password)
1  {
2      if (password.length() < MinimumPasswordLength)
3      {
4          throw logic_error("Password is too short");
5      }
6
7      string encrypted;
8      // work
9      return encrypted;
10 }
```

```
0  std::string encryptPassword(const std::string& password)
1  {
2      // import std and check length as above
3      string encrypted;
4
5      // default-construct encrypted
6      encrypted = password;
7
8      // assign to encrypted
9      encrypt(encrypted);
```

```
10
11      return encrypted;
12  }
```

```
1  std::string encryptPassword(const std::string& password)
2  {
3      // define and initialize via copy
4      // constructor
5      string encrypted(password);
6
7      encrypt(encrypted);
8      return encrypted;
9  }
```

This might seem intuitive at first, but there are real benefits to postponing your declarations and initialization *as much as possible*. Because we are dealing with the overhead of constructing and destructing every object in our scope, being able to *"tighten-up"* our program can lead to better optimizations. We can fall in the cases, where we might think we are using a certain variable, because our compiler is not complaining about it, but take for instance the examples. We can see that the construction of the string object happens before the test case. If the `try-catch` block throws, the string was created for no apparent reason. If we refactor the code, it will create the string only when it is really needed, instead of preemptively allocating it. We can even further improve this code by assigning it a value as we are defining it. If we first define it and then assign a value, we are calling the default constructor first, and then we are calling the assignment operator to give the string a value. But if we just make this into one whole statement it will perform only one operation. We know that copy constructors are sometimes even *more efficient* than the default ones, so we can utilize this to our advantage. So we can not only postpone definitions of a variable, but we can also postpone initialization operations too, where we can optimize how a variable is created. This way we are avoiding calls to the constructor and destructor and sparing ourselves unneeded objects.

```
0  // Approach A: define outside loop
1  Widget w;
2  for (int i = 0; i < n; ++i)
3  {
4      w = some value dependent on i;
5  }
6
7  // Approach B: define inside loop
8  for (int i = 0; i < n; ++i)
9  {
10     Widget w( some value dependent on i);
11 }
```

But what about this case, where we have loops. Should we define the variable outside of the scope of the loop or initialize it *every* time we go over the iterations. We have two situations. In case A : we have `1 constructor + 1 destructor + n assignments`. In case B : we have `n constructors + n destructors`. Case A is valid when the assignment operator is *more efficient* than the constructor,

which can be seen in situations where `n` is very large. Otherwise approach *B* is better. Also keep in mind that case *A* reveals the variable object into *larger scope* than we need it to be. So in general, we should rely on the second case, where the object is assigned inside of the loop body.

Resources

Memory Management

- Why use new and delete
- What does delete do
- 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

Pointers

- Circular references
- What is reference counting
- Smart pointer
- CPP and smart pointers
- Ownership semantics
- What is a smart pointer
- Shared pointer
- Unique pointer
- Weak pointer
- Reference counting with smart pointers