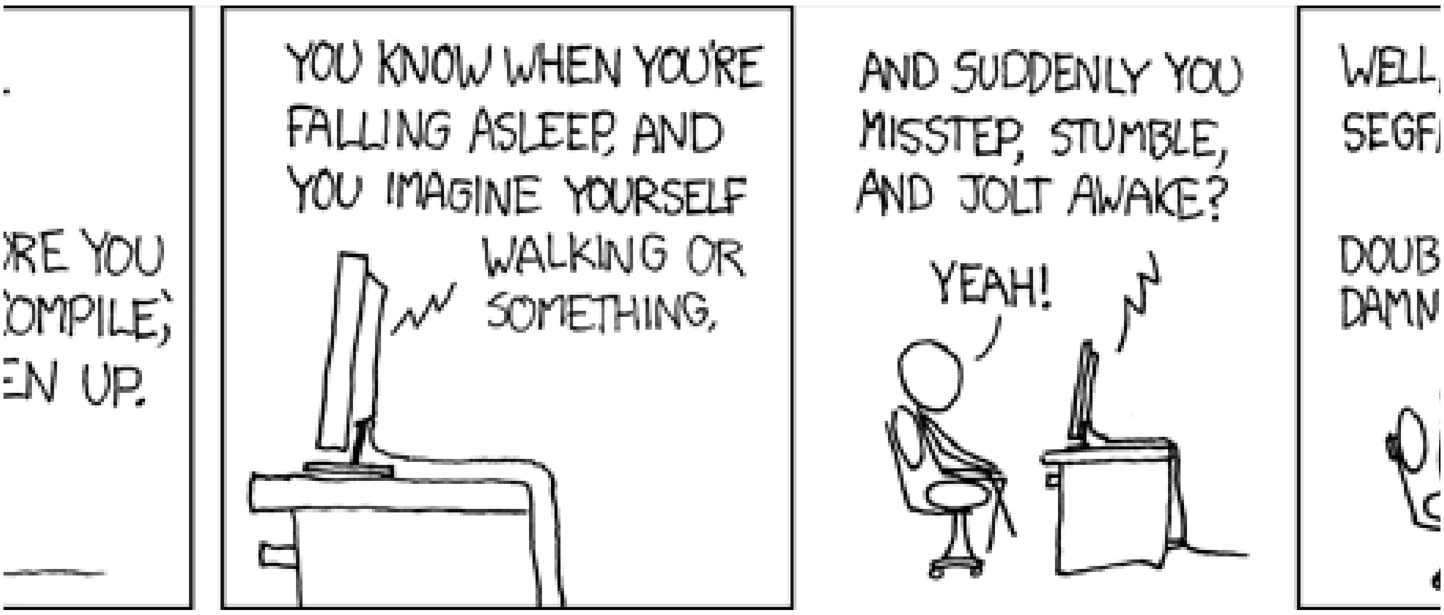
DEV Create account


RE YOU
OMPILE,
EN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, SEGF
DOUB DAMN

10x learner
Posted on 22 Jan 2020 • Originally published at 10xlearner.com on 22 Jan 2020

💖 8

Memory Management and RAII

#tipsandtricks #c #raii #memorymanagement

Hi dear reader, I’m Xavier Jouvenot and in this article, we are going to talk about Memory Management, more specifically in C++, even if the concept can be extended to other languages. This post was inspired by a rule from the first chapter of Code Craft, by Pete Goodliffe, on Defensive programming

Why should you even care about memory management ?

This is a fair question.We are 2020, some languages have tried several way to remove this task away from the programmer. For example, some languages don’t let the possibility to the programmer to dynamically allocate memory, while some other have integrated a garbage collector system which is supposed to do that for us, isn’t it ? 🤔

Sadly, even a garbage collector can’t magically handle all memory management, and even if you restrict yourself to language without dynamic memory allocation, you should understand how static memory allocation works for this language, to use it more intelligently. Whatever the case, it is important for you to understand how the

In C and C++, as long as you don't use pointers and dynamic allocations, you are good, the memory is handle for you by the Stack. But when you will use (and you will probably have to use), dynamic allocation, then, you should be careful about how you manage you memory to avoid corrupting your memory, using invalid memory, memory leaks, and all other problems related to dynamic allocation. 😮

Old school memory management

In C and before C++11, the only way to use memory allocation available were to use the keywords `malloc`,or `new`, and `free` ,or `delete`.The first are for allocating dynamic memory space, taking the ownership on some memory, and the seconds one are for freeing allocated memory, releasing the memory on which we had ownership.

Those methods gave and still give to the developer to do whatever they want about memory management, so the responsibility is completely in the developer hands. If use badly, you could end up trying to free a memory block that was yours or forget to free some memory blocks, and end up with some weird undefined behavior. This made the task difficult for many programmers, so much that many developers are still afraid of doing memory management.

But some developers reflected on it and came up with techniques and principles that could allow us to use dynamic allocation without risks, and this is what we are going to see now 😉

Resource Allocation Is Initialization

RAII , as the title said, stands for *"Resource Allocation Is Initialization"*.This is a principle which state that when allocating some memory, you should define its lifetime.

For stack-allocated objects, the RAII is granted by the system.Each variable/object will be deleted at the end of the scope in which it was created.But for heap-allocated objects, RAII is not granted by the system.So you have to make sure that every object dynamically allocated will be destroyed at the end of the scope in which it was created.This requires some discipline, but avoid a lot of pain when you have to find the origin of a memory leak with tools like valgrind.

And if you still use older C++, you can extract the concept of those smart pointers to integrate similar objects in your code base. 😉

std::unique_ptr

The first smart-pointer, and probably the most used, is `std::unique_ptr`.This smart pointer owns a pointer to a dynamically allocated element, and free the memory dynamically when it's destroyed.Let's take an example

```
{
    // some instructions
    std::unique_ptr<int> smart(new int(4));
    // some other operations
}
```

Here you have some dynamic allocation happening with the `new int(4)`, and it is encapsulate by the smart pointer so that, at the end of scope, the memory dynamically allocated in the variable `smart` will be free automatically. No need to add a `delete` keyword, so add too much or too little of them.It also work as a member of an object:

```
struct MyClass{
    MyClass(float value) : smart(new float(value)) {}

    std::unique_ptr<float> smart;
};

//....
{
    MyClass objectWithSmartPointer (5.0f);
}
```

destroyed at the end of the scope, then, the memory allocated dynamically in the member of the object will also be free automatically, no need to add any delete in the class destructor ! Pretty cool, isn't it ?! 😄

To make the thing even easier for the developer, in C++14, the function std::make_unique has been added, which allows use to avoid unnecessary copy during the creation of an element in the heap **AND** , when combined with the keyword `auto` makes code even simpler to read and write!

```
{
    // some instructions
    auto smart = std::make_unique<int>(4);
    // some other operations
}
```

Here, even the keyword `new` has disappeared ! And the variable `smart` is a `std::unique_ptr` ready to be used. 🙂
So now, you code can be free from `new` and `delete` keywords ! 🍾

But, even if the `unique_ptr` is a game changer in memory management, this isn't a silver bullet either! It has its own limitation !Indeed, since it has the ownership on the memory allocated by it, it is difficult to have some memory shared between objects. Luckily for us, the next smart pointer will help us to do just that. 😉

std::shared_ptr

Like `std::unique_ptr`, `std::shared_ptr` is a smart pointer, but unlike `std::unique_ptr` which has the ownership over the object it points at, `std::shared_ptr` has a **shared** ownership over the object it points at.This mean that you can have several `std::shared_ptr` owning the same object whereas if you try to do that with `std::unique_ptr`, the first `std::unique_ptr` to be destroy will destroy the object pointed, and the other will point to not-owned memory which will trigger undefined behavior in your program.

destroyed.Moreover, as for the `std::unique_ptr` smart pointer you have the function template `std::make_unique`, you have the function template [std::make_shared] which does the same for `std::shared_ptr`.

```
auto smart = std::make_shared<int>(4);
auto otherSmart = smart;
```

Using only `std::shared_ptr`, you will probably encounter one last case in your program.For now, we have been talking about the case where are the pointers want to have the ownership.But what if you what to have some pointers with the ownership over an object and some without any ownership to the same object ? 🤔

I suppose you could use raw pointers, as they are not owning anything and check all the time if the element pointed is still valid somehow unless the `std::shared_ptr` are all destroy between your check and when you want to use the object pointed. You can imagine this case for a connection to a service where one entity in your program as the ownership, and the other part of you program only want to access the connection if it is valid, for example.

To facilitate such situation, there is a last smart pointer, which is non-owning, named std::weak_ptr. This smart pointer has a reference to an object owned by at least one `std::shared_ptr`, and can take the shared ownership if you want to, so that you won’t have a problem if when accessing the object pointed, the other `std::shared_ptr` are destroyed.

I encourage you too look at examples online, and play with it on godbolt to master how `std::shared_ptr` and `std::weak_ptr` work together 😉

What about "regular" pointer then ?

Now that we have smart pointers, we can ask ourselves if there is a need for regular raw pointers.Actually, there are still some use for it. If you want to use inheritance and dynamic_cast, for example, then you may want to use raw pointers.

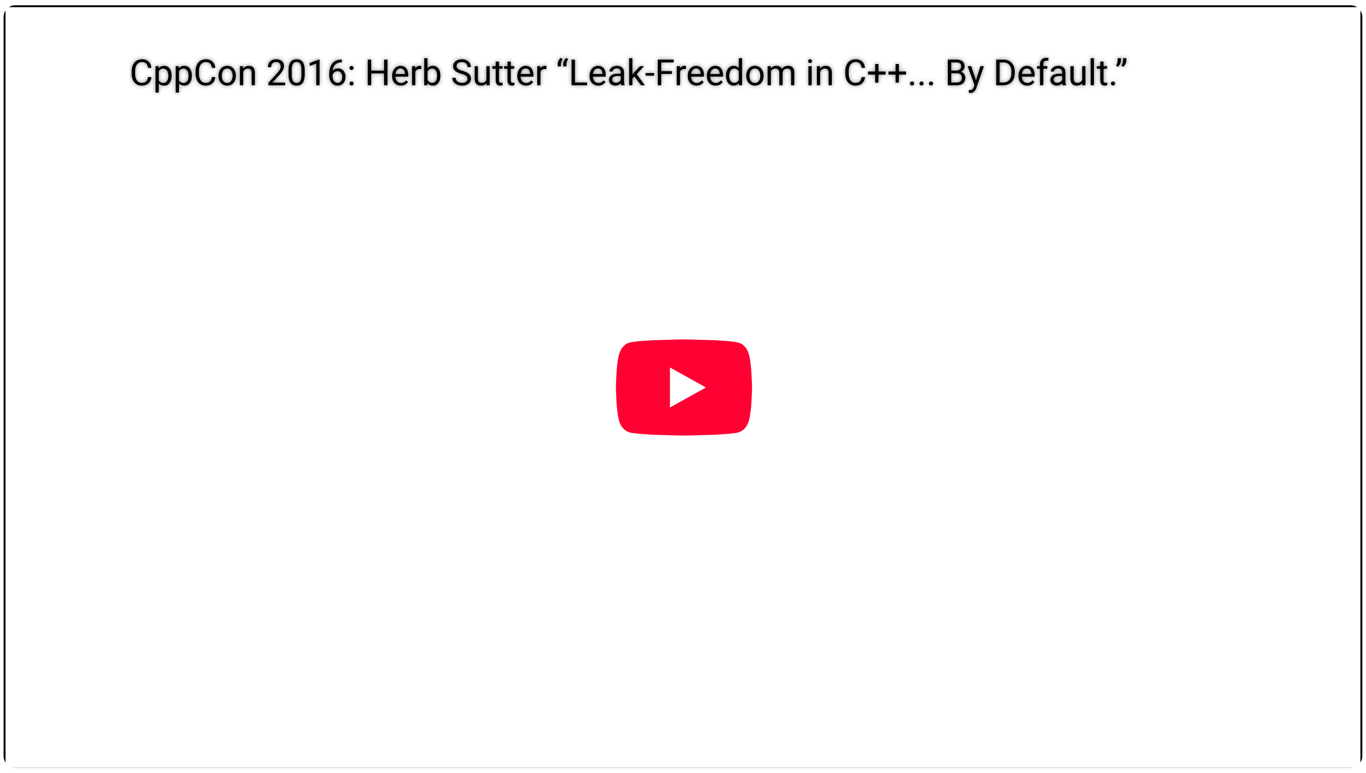
could or not be there, but now, you should prefer using std::optional.

Conclusion

If I have to resume the point of this article in 4 words, it would be "Please, use smart pointers" 😆

More seriously, every programmer should have a notion of how memory is managed by the language he uses, so that we would understand what is going when the program has a problem with memory management. And more especially C++ developer, since all those tools, the smart pointers are available to us, we should definitely use the one that match perfectly our needs in each situation.

Thank you all for reading this article ! If you want more information, I recommend you with the talk of Herb Sutter named "Leak Freedom in C++... By Default"And until my next article, have an splendid day 😉


CppCon 2016: Herb Sutter “Leak-Freedom in C++... By Default.”