28 October 2019

Lazy Initialisation in C++

See my new website at cppstories.com

Lazy initialisation is one of those design patterns which is in use in almost all programming languages. Its goal is to move the object's construction forward in time. It's especially handy when the creation of the object is expensive, and you want to defer it as late as possible, or even skip entirely.

Keep reading and see how you can use this pattern with the C++ Standard Library.

Table of Contents

Update: Read the next article about Lazy init and Multi-threading.

This article is a guest post from **Mariusz Jaskółka**:

Recently Published

C++ Stories in March

Popular Posts:

Using C++17 std::optional
Everything You Need to Know About std::variant f

> ***Mariusz*** *is a professional programmer for whose C++ is both his job and passion. He works for Novomatic Technologies Poland and writes texts to Polish blog cpp-polska. Besides programming he loves mushroom picking, mountain hikes and everything music-related. Check out his github and linkedin profiles.*

Originally published in Polish at cpp-polska.pl

Problem description

Let's make a real-life example. We have an RAII object that represents a file on the hard drive. We deliberately won't use `std::ifstream` class, as it allows late file opening so that using *late initialisation* pattern would be pointless.

Consider the following class:

```
class File
{
public:
    File(std::string_view fileName)
    {
        std::cout << "Opening file " << fileNa
    }
    ~File()
    {
        std::cout << "Closing file" << std::en
    }
    File(const File&) = delete;
    File(File&&) = default;
    File& operator=(const File&) = delete;
    File& operator=(File&&) = default;

    void write(std::string_view str)
    {
        std::cout << "Writing to file: " << st
    }
};
```

As we can see, the file is opened in the constructor, and that's the only place we can do it.

We can use such class to save a configuration file:

```
class Config
{
```

How to Iterate Through Directories in C++
Vector of Objects vs Vector of Pointers And Memo
In-class Member Initialisation: From C++11 to C++
Notes on C++ SFINAE, Modern C++ and C++20 Conc
Runtime Polymorphism with std::variant and std::

```
    File file;
public:
    Config() : file{"config.txt"}
    {
        std::cout << "Config object created" <
    }

    void addOption(std::string_view name, std:
    {
        file.write(name);
        file.write(" = ");
        file.write(value);
        file.write("\n");
    }
};
```

Here’s a simple usage:

```
int main()
{
    Config c;
    std::cout << "Some operations..." << std::
    c.addOption("dark_mode", "true");
    c.addOption("font", "DejaVu Sans Mono");
}
```

Run on Wandbox

The problem with this implementation is that we presumably open the file long time before we really need to write to it. This may block other processes from manipulating this file, which is an undesirable side effect. We would instead open the file when the first call to `addOption` function occurs.

We can achieve such behaviour in several ways. Let’s have a look.

The First Way - Uninitialized Raw Pointer:

Pointers seem to be the solution at first glance - they can point to some value or to “nothing” (`nullptr`). Let’s go back to the example and then discuss why this is rather a bad idea.

```
class Config
{
    File* file{nullptr};

public:
    Config()
    {
        std::cout << "Config object created" <
    }

    ~Config()
    {
        delete file;
    }

    // ah... need to implement rule of 5...7 r

    void addOption(std::string_view name, std:
    {
        if (!file)
            file = new File{"config.txt"};
        file->write(name);
        file->write(" = ");
        file->write(value);
        file->write("\n");
    }
};
```

Run on Wandbox

In modern C++, holding allocated memory on the heap, under a raw pointer is considered to be a bad idea in most scenarios. First of all, mixing them with the exception mechanism can lead us to memory leaks. They also require manual memory freeing, which can be bypassed using the handy and lightweight RAII design pattern.

If we declared a destructor it also means we have to follow the Rule of Five and implement copy ctor, assignment op and move semantics for the class.

The Second Way – Smart Pointer

Having a smart pointer can free us from extra boilerplate code:

```
class Config
{
    std::unique_ptr<File> file{};
public:
    Config()
    {
        std::cout << "Config object created" <
    }

    void addOption(std::string_view name, std:
    {
        if (!file)
            file = std::make_unique<File>("con
        file->write(name);
        file->write(" = ");
        file->write(value);
        file->write("\n");
    }
};
```

Run on Wandbox

Our problem is solved in a much more elegant way. Compared to the original implementation, this method has one drawback though - the object is allocated on the heap. Allocation on the heap requires a system call (*syscall*), and the number of system calls should be rather minimized. Using objects from under the pointer might also cause less possibility of program optimization compared to objects referenced from the stack. That can lead us to another solution ...

The Third Way – `std::optional` (C++17)

```
class Config
{
    std::optional<File> file{};
public:
    Config()
    {
        std::cout << "Config object created" <
    }

    void addOption(std::string_view name, std:
    {
        if (!file)
            file.emplace("config.txt");
        file->write(name);
        file->write(" = ");
        file->write(value);
        file->write("\n");
    }
};
```

Run on Wandbox

We can notice that the above code doesn't differ very much with the previous one. The `unique_ptr` and `optional` references are similar, but the implementation and purpose of those classes vary significantly. First of all, in the case of `std::optional` our objects memory is on the stack.

It is worth mentioning that if you are not using C++17, but some older language version, you can use the Boost.Optional library, which implements the almost identical class.

(Smart) Pointers vs `std::optional`

- `unique_ptr` is – as the name implies – a wrapper around the raw pointer, while `optional` object contains memory required to its reservation as a part of the object.
- **Default constructor** of `unique_ptr` class just sets the underlying pointer to `nullptr`, while `optional` object allocation also allocates (on the stack) memory for an underlying object.
- **make_unique** helper function does two things – it reserves memory required to object construction on the heap, and after that, it constructs an object using that memory. Its behaviour can be compared to the ordinary *new operator*. On the other hand, the member function `optional::emplace`, which could be considered as an equivalent, only calls object construction with the usage of on-stack preallocated memory – so it works like less known *placement new operator*.

The consequences of the above features are:

- **Copy constructor** of `unique_ptr` doesn't exist. We can use another smart pointer - `shared_ptr` so that we could copy the pointer, but it would still point one object on the heap. The `optional` class, on the other hand, invokes deep copy of the underlying object when is copied itself. The situation is similar in the case of the operator `=`.

- **Move constructor** of `unique_ptr` class doesn't invoke deep copy either. It just moves underlying object management to a different instance. The `optional` class invokes underlying object move constructor.
- **Destructor** of `unique_ptr` class not only destroys underlying object (calls destructor of it), but also frees memory occupied by it – so it works exactly like `operator delete`. `optional`'s destructor calls underlying object's destructor, but it doesn't have to free any memory – it will be available to next objects appearing on the stack.

Which Option Should I Use?

The use of the `optional` class described earlier may not be the first that comes to mind those who use it. Instead, it is a class that expresses that an object *is present* or *is not*. Here we revealed the fact that the object *does not yet exist, but it will probably be in the future*. This is, however, perfectly valid usage of this class.

The answer to the question "which method should I use to express late initialisation?" isn't that trivial though. I would advise beginners to use `optional` by default (form *std* or *boost*). However, if we examine this issue in more detail, we can draw the following conclusions:

- **Smart pointers** should be used mainly when we want to postpone the reservation of some large amount of memory, e.g. intended for storing the contents of an image file.
- `std::optional` should be preferred when not the memory (its amount) is essential, but the reservation of other types of resources (such as file handles, network sockets, threads, processes). It is also worth using it when the construction of the object is not possible immediately but depends on some parameter whose value is not yet known. In addition, using this class will usually be more efficient - especially if we have, for example, a large vector of such objects and want to iterate over them.

We also cannot forget about the properties of the described classes, especially about how they are copied and moved.

Update: Read the next article about Lazy init and Multi-threading.

Back to you:

- Do you use some form of lazy initialisation?
- What techniques do you use to implement it?
- Maybe you have some good example?

Let us know in comments

Recent Posts

C++ Stories in March
C++ Lambda Story in Print @CppStories
First Year At Patreon
C++ at the end of 2020! @CppStories
One Trick with Private Names and Function Templates @C++Stories

Newer Post Home Older Post