The Standard Library Changes

This section delves deep into the std::optional utility introduced in C++17.

WE'LL COVER THE FOLLOWING

Introduction

While new language features allow you to write more compact code, you also need the tools - in the form of the Standard Library types. The classes and systems that you can find in the Library can significantly enhance your productivity. C++17 offers even more handy instruments: for example the filesystem, new vocabulary types, and even parallel algorithms! We'll now look at a prominent new feature called `std::optional`.

Nullable Types

One approach is to achieve "null-ability" by using unique values (`-1`, infinity, `nullptr`). Before use, you need to compare the object against the predefined value to see if it's not empty. Such a pattern is widespread in programming. For instance `string::find` returns a value that represents the position or `npos` when it's "null" or the pattern is not found.

Alternatively, you could try with `std::unique_ptr<Type>` and treat the empty pointer as not initialised. That works but comes with the cost of allocating memory for the object and is not a recommended technique.

Another technique is to build a wrapper that adds a boolean flag to other types. Such wrapper can quickly determine the state of the object. And this is how in a nutshell works `std::optional`.

Optional types that come from the functional programming world bring type safety and expressive- ness. Most other languages have something similar: for example `std::option` in Rust, `Optional<T>` in Java, `Data.Maybe` in Haskell.

std::optional

`std::optional` was added in C++17 and brings a lot of experience from boost::optional that has been available for many years. With C++17 you can just #include and use the type.

> `std::optional` was available also in Library Fundamentals TS, so there's a chance that your C++14 compiler could also support it in the `<experimental/optional>` header.

`std::optional` is still a value type (so it can be copied, via deep copy). Additionally, `std::optional` doesn't need to allocate any memory on the free store.

`std::optional` is a part of C++ **vocabulary types** along with `std::any`, `std::variant` and `std::string_view`.

When To Use

You can usually use an optional wrapper in the following scenarios:

If You Want to Represent a Nullable Type

- Rather than using unique values (like `-1`, `nullptr`, `NO_VALUE` or something)
- For example, a user's middle name is optional. You could assume that an empty string would work here, but knowing if a user entered something or not might be important. `std::optional<std::string>` gives you more

information.

Return a Result of Some Computation (Processing) That Fails to Produce a Value and Is Not an Error

For example, finding an element in a dictionary: if there's no element under a key, it's not an error, but we need to handle the situation.

To Perform Lazy-Loading of Resources

For example, if the construction of a resource type is substantial, or if there's no default constructor, you can define it as `std::optional<Resource>`. In that form, you can pass it around the system, and then initialise it (load a resource), when the application wants to access it for the first time.

To Pass Optional Parameters into Functions

The documentation for `boost.optional` has a useful summary on when we should use the type, see in When to use Optional:

> It is recommended to use `optional<T>` in situations where there is exactly one, clear (to all parties) reason for having no value of type `T`, and where the lack of value is as natural as having any regular value of `T`.

While sometimes the decision to use optional might be blurry, it best suits the cases when the value is empty, and it's a normal state of the program.

If this is a lot to take in, don't worry. We'll look at an example of how `std::optional` actually works.