The Basics

The basics talk about the former kind of exceptions thrown, and when to use std::any.

In `C++14` there weren't many options for holding variable types in a variable. You could use `void*`, of course, but this wasn't safe. `void*` is just a raw pointer, and you have to manage the whole object lifetime and protect it from casting to a different type.

Potentially, `void*` could be wrapped in a class with some type discriminator.

```
class MyAny
{
    void* _value;
    TypeInfo _typeInfo;
};
```

Making Type-Safe

As you see, we have some basic form of the type, but there's a bit of coding required to make sure `MyAny` is type-safe. That's why it's best to use the Standard Library rather than rolling a custom implementation.

And this is what `std::any` from C++17 is in its basic form. It lets you store anything in an object, and it reports errors (or throw exceptions) when you'd like to access a type that is not active.

A little demo:

```
#include <string>
#include <iostream>
```

```
#include <any>
#include <map>
using namespace std;


int main()
{
    auto a = std::any(12);

    // set any value:
    a = std::string("Hello!");
    a = 16;
    // reading a value:

    // we can read it as int
    std::cout << std::any_cast<int>(a) << '\n';

    // but not as string:
    try
    {
        std::cout << std::any_cast<std::string>(a) << '\n';
    }
    catch(const std::bad_any_cast& e)
    {
        std::cout << e.what() << '\n';
    }

    // reset and check if it contains any value:
    a.reset();
    if (!a.has_value())
    {
        std::cout << "a is empty!" << '\n';
    }

    // you can use it in a container:
    std::map<std::string, std::any> m;
    m["integer"] = 10;
    m["string"] = std::string("Hello World");
    m["float"] = 1.0f;

    for (auto &[key, val] : m)
    {
        if (val.type() == typeid(int))
            std::cout << "int: " << std::any_cast<int>(val) << '\n';
        else if (val.type() == typeid(std::string))
            std::cout << "string: " << std::any_cast<std::string>(val) << '\n';
        else if (val.type() == typeid(float))
            std::cout << "float: " << std::any_cast<float>(val) << '\n';
    }
}
```

The example above shows us several things:

- `std::any` is not a template class like `std::optional` or `std::variant`.
- by default it contains no value, and you can check it via `.has_value()`.

- you can reset an `any` object via `.reset()`.
- it works on "decayed" types - so before assignment, initialisation, or emplacement the type is transformed by std::decay.
- when a different type is assigned, then the active type is destroyed.
- you can access the value by using `std::any_cast<T>`. It will throw `bad_any_cast` if the active type is not `T`.
- you can discover the active type by using `.type()` that returns `std::type_info` of the type.

When to Use

While `void*` might be an extremely unsafe pattern with some limited use cases, `std::any` adds type-safety, and that's why it has more applications.

Some possibilities:

- In Libraries - when a library type has to hold or pass anything without knowing the set of available types
- Parsing files - if you really cannot specify what the supported types are
- Message passing
- Bindings with a scripting language
- Implementing an interpreter for a scripting language
- User Interface - controls might hold anything
- Entities in an editor

In many cases, you can limit the number of supported types, and that's why `std::variant` might be a better choice. Of course, it gets tricky when you implement a library without knowing the final applications - so you don't know the possible types that will be stored in an object.

Let's move to low-level details like object creation in the next lesson!