Object Creation

The different methods of in-place object creation are discussed briefly in this lesson.

WE'LL COVER THE FOLLOWING

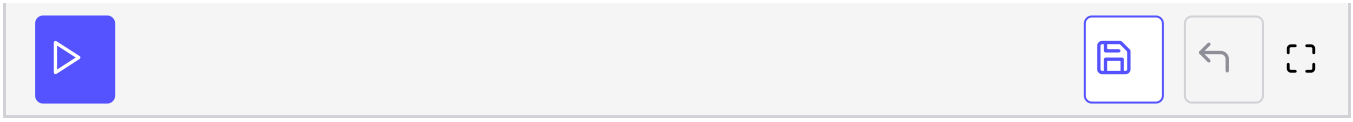
There are several ways you can create `std::any` object:

- a default initialisation - then the object is empty
- a direct initialisation with a value/object
- in place `std::in_place_type`
- via `std::make_any`

You can see it in the following example:

```
#include <string>
#include <iostream>
#include <any>
#include <cassert>
using namespace std;

class MyType
{
  int a, b;

  public:
    MyType(int x, int y) : a(x), b(y) { }
};

int main()
{
    // default initialization:
    std::any a;
```

```
    assert(!a.has_value());

    // initialization with an object:
    std::any a2{10}; // int
    std::cout <<  "a2 is: " << std::any_cast<int>(a2) << '\n';
    std::any a3{MyType{10, 11}};

    // in_place:
    std::any a4{std::in_place_type<MyType>, 10, 11};
    std::any a5{std::in_place_type<std::string>, "Hello World"};
    std::cout <<  "a5 is: " << std::any_cast<std::string>(a5) << '\n';

    // make_any
    std::any a6 = std::make_any<std::string>("Hello World");
    std::cout <<  "a6 is: " << std::any_cast<std::string>(a6) << '\n';
}
```



In Place Construction

Following the style of `std::optional` and `std::variant`, `std::any` can use `std::in_place_- type` to efficiently create objects in place.

Complex Types

In the below example a temporary object will be needed:

```
std::any a{UserName{"hello"}};
```

but with:

```
 std::any a{std::in_place_type<UserName>, "hello"};
```

The object is created in place with the given set of arguments.

std::make_any

For convenience `std::any` has a factory function called `std::make_any` that returns

```
return std::any(std::in_place_type<T>, std::forward<Args>(args)...);
```

So in the previous example we could also write:

```
auto a = std::make_any<UserName>{"hello"};
```

`make_any` is probably more straightforward to use.

Changing the Value

When you want to change the currently stored value in `std::any` then you have two options: use `emplace` or the assignment:

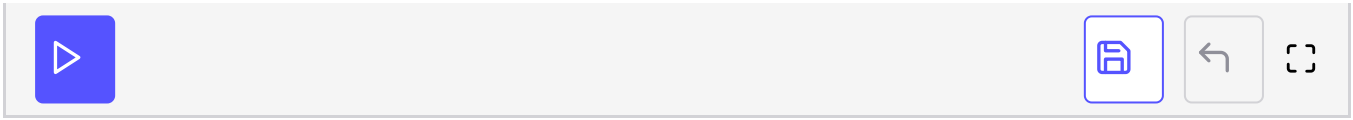
```
class MyType {
    int a, b;

public:
    MyType(int x, int y) : a(x), b(y) { }
};

int main() {
    // default initialization:
    std::any a;
    assert(!a.has_value());

    a = MyType(10, 11);
    a = std::string("Hello");

    a.emplace<float>(100.5f);
    a.emplace<std::vector<int>>({10, 11, 12, 13});
    a.emplace<MyType>(10, 11);

    return 0;
}
```

Object Lifetime

The crucial part of being safe for `std::any` is not to leak any resources. To achieve this behaviour `std::any` will destroy any active object before assigning a new value.

```
void* operator new(std::size_t count) {
    std::cout << "allocating: " << count << " bytes" << std::endl;
    return malloc(count);
}

void operator delete(void* ptr) noexcept {
    std::puts("global op delete called");
    std::free(ptr);
}

class MyType {
```

```
public:
    MyType() { std::cout << "MyType::MyType\n"; }
    ~MyType() { std::cout << "MyType::~MyType\n"; }
};

int main() {
    {
        std::any var = std::make_any<MyType>();
        var = 100.0f;
        std::cout << std::any_cast<float>(var) << '\n';
    }
}
```



If the constructors and destructors were instrumented with prints, we would get the following output:

```
MyType::MyType
MyType::~MyType
100
```

The `any` object is initialised with `MyType`, but before it gets a new value (of `100.0f`) it calls the destructor of `MyType`.

That's all for the object creation. We'll look into different ways of accessing the stored values in the next lesson.