Shared Pointers

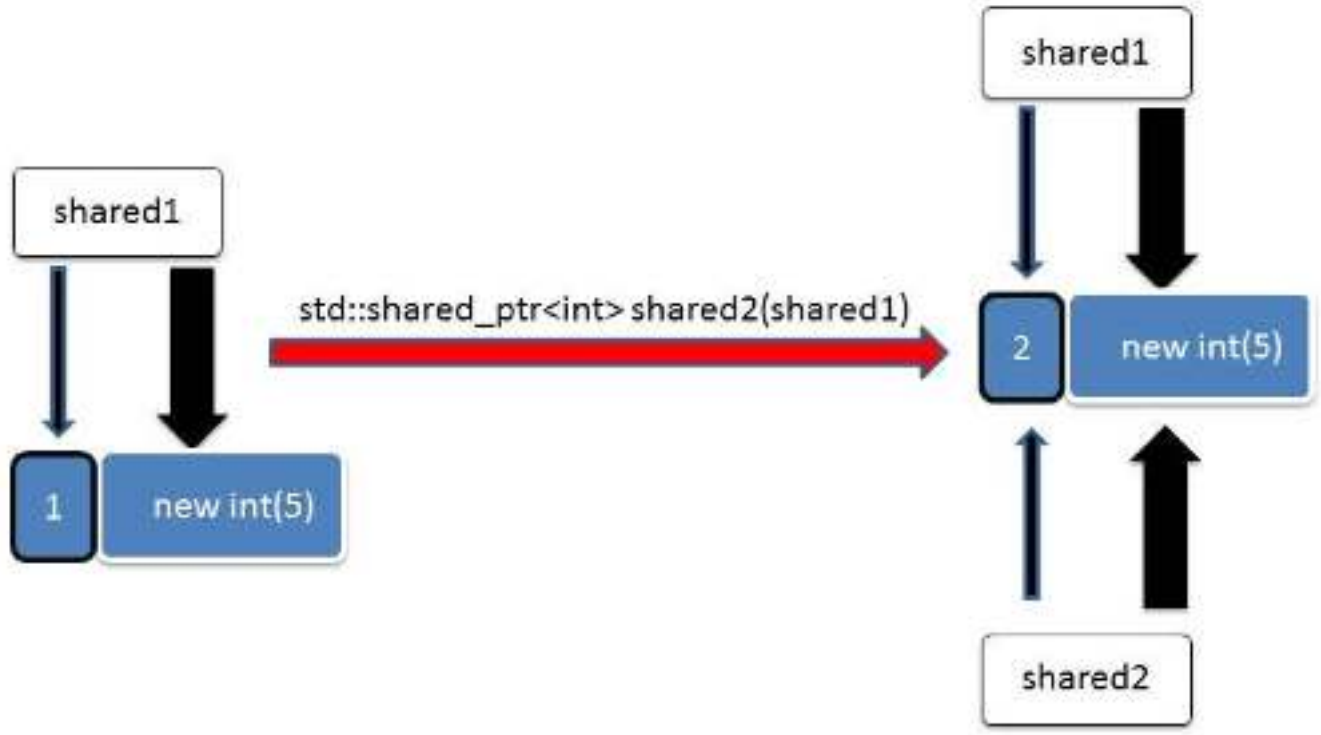
Next, we will go over shared pointers, which follow the principle of keeping a reference count to maintain the count of its copies. The lesson below elaborates further.

WE'LL COVER THE FOLLOWING

Introduction

std::shared_ptr shares ownership of the resource. They have two handles: one for the resource, and one for the reference counter. By copying an `std::shared_ptr`, the reference count is increased by one. It is decreased by one if the `std::shared_ptr` goes out of scope. If the reference counter becomes the value 0, the C++ runtime automatically releases the resource, since there is no longer an `std::shared_ptr` referencing the resource. The release of the resource occurs exactly when the last `std::shared_ptr` goes out of scope. The C++ runtime guarantees that the call of the reference counter is an atomic operation. Due to this management, `std::shared_ptr` consumes more time and memory than a raw pointer or `std::unique_ptr`.

Take a look at the image below to better visualize the concept.



Due to `shared1`, `shared2` is initialized. In the end, the reference count is 2 and both smart pointers have the same resource.

Methods

In the following table, we will see the methods of `std::shared_ptr`.

Name	Description
`get`	Returns a pointer to the resource.
`get_deleter`	Returns the delete function.
`reset`	Resets the resource.
`swap`	Swaps the resources.
`unique`	Checks if the `std::shared_ptr` is the exclusive owner of the resource.

`use_count`	Returns the value of the reference counter.

Methods of std::shared_ptr

std::make_shared

The helper function std::make_shared creates the resource and returns it in an `std::shared_ptr`. Use `std::make_shared` rather than directly creating an `std::shared_ptr` because `std::make_shared` is much faster. Additionally, such as in the case of `std::make_unique`, `std::make_shared` guarantees no memory leaks.

std::shared_ptr from this

> This unique technique, in which a class derives from a class template having itself as a parameter, is called CRTP and stands for **C**uriously **R**ecurring **T**emplate **P**attern.

Using the class std::enable_shared_from_this, we can create objects that return an `std::shared_ptr` to themselves. To do so, we must publicly derive the class from `std::enable_shared_from_this`. So the class `ShareMe` support the method `shared_from_this`, and return `std::shared_ptr`:

```
// enableShared.cpp

#include <iostream>
#include <memory>

class ShareMe: public std::enable_shared_from_this<ShareMe>{
public:
  std::shared_ptr<ShareMe> getShared(){
    return shared_from_this();
  }
};

int main(){

  std::cout << std::endl;

  std::shared_ptr<ShareMe> shareMe(new ShareMe);
  std::shared_ptr<ShareMe> shareMe1= shareMe->getShared();
  {
    auto shareMe2(shareMe1);
    std::cout << "shareMe.use_count(): "  << shareMe.use_count() << std::endl;
```

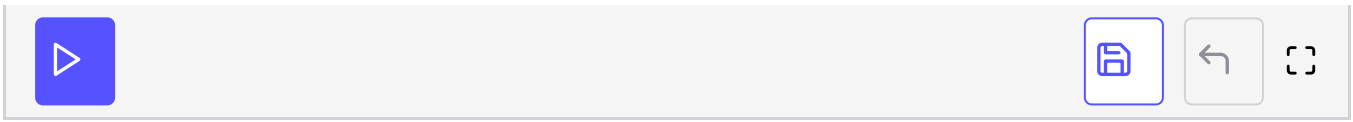
```
  }
  std::cout << "shareMe.use_count(): "  << shareMe.use_count() << std::endl;


  shareMe1.reset();

  std::cout << "shareMe.use_count(): "  << shareMe.use_count() << std::endl;

  std::cout << std::endl;

}
```



The smart pointer `shareMe` (line 17) is copied by `shareMe1` (line 18) and `shareMe2` (line 20), and all of them

- reference the very same resource.
- increment and decrement the reference counter.

The call `shareMe->getShared()` in line 18 creates a new smart pointer. `getShared()` (line 9) internally uses the function `shared_from_this`.

Further information

- std::shared_ptr
- std::make_shared
- CRTP
- std::enable_shared_from_this

The examples in the next lesson will build on our understanding of this topic.