# Shared Pointers

Next, we have the shared pointer. It follows the principle of keeping a reference count to maintain the count of its copies. The lesson below elaborates further.

WE'LL COVER THE FOLLOWING

- std::make_shared
- std::shared_ptr from this

std::shared_ptr shares the ownership of the resource. They have two handles. One for the resource and one for the reference counter. By copying a `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 and therefore there is no `std::shared_ptr` referencing the resource, the C++ runtime automatically releases the resource. The release of the resource takes place at exactly the time at which the last `std::shared_ptr` goes out of scope. The C++ runtime guarantees that the call of the reference counter is an atomic operation. Because of this management, `std::shared_ptr` uses more time and memory than a raw pointer or `std::unique_ptr`.

In the following table are 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`**

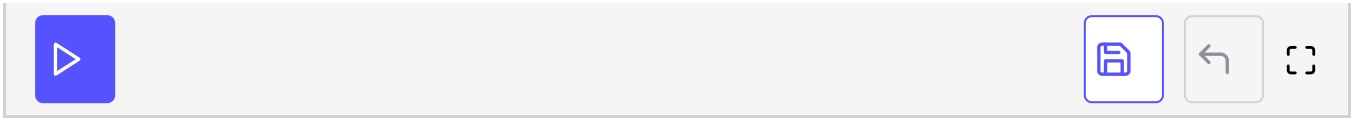
## std::make_shared #

The helper function std::make_shared creates the resource and returns it in a `std::shared_ptr`. You should use `std::make_shared` instead of the direct creation of a `std::shared_ptr`, because `std::make_shared` is a lot faster.

The following code sample shows a typical use case of a `std::shared_ptr`.

```
// sharedPtr.cpp
#include <iostream>
#include <memory>

class MyInt{
public:
  MyInt(int v):val(v){
    std::cout << "Hello: " << val << std::endl;
  }
  ~MyInt(){
    std::cout << "Good Bye: " << val << std::endl;
  }
private:
  int val;
};

int main(){
  auto sharPtr= std::make_shared<MyInt>(1998);        // Hello: 1998
  std::cout << sharPtr.use_count() << std::endl;      // 1

  {
    std::shared_ptr<MyInt> locSharPtr(sharPtr);
    std::cout << locSharPtr.use_count() << std::endl; // 2
  }
  std::cout << sharPtr.use_count() << std::endl;      // 1

  std::shared_ptr<MyInt> globSharPtr= sharPtr;
  std::cout << sharPtr.use_count() << std::endl;      // 2

  globSharPtr.reset();
  std::cout << sharPtr.use_count() << std::endl;     // 1
  sharPtr= std::shared_ptr<MyInt>(new MyInt(2011));  // Hello:2011
  // Good Bye: 1998
```

```
  // Good Bye: 1998
  // Good Bye: 2011
  return 0;
}
```



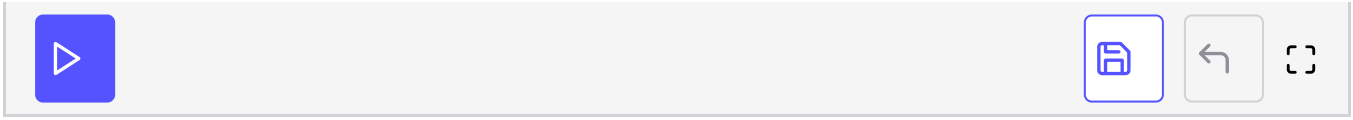
`std::shared_ptr`

The callable is in this example a function object. Therefore you can easily count how many instances of a class were created. The result is in the static variable `count`.

## std::shared_ptr from this #

With the class std::enable_shared_from_this, you can create objects which return a `std::shared_ptr` on itself. For that you have to derive the class public from `std::enable_shared_from_this`. So the class support the method `shared_from_this` to return `std::shared_ptr` to this:

```
// 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::shared_ptr<ShareMe> shareMe(new ShareMe);
  std::shared_ptr<ShareMe> shareMe1= shareMe->getShared();

  std::cout << (void*)shareMe.get() << std::endl;   // 0x152d010
  std::cout << (void*)shareMe1.get() << std::endl;  // 0x152d010
  std::cout << shareMe.use_count() << std::endl;    // 2

  return 0;
}
```



`std::shared_ptr` from this"

You can see in the code sample that the `get` methods reference the same object.

Now, let’s dive into weak pointers.