Unique Pointers

The first type of smart pointer in this section is the unique pointer. It limits the access to its resource, thereby, maintaining its privacy.

WE'LL COVER THE FOLLOWING

std::unique_ptr exclusively takes care of its resource. It automatically releases the resource if it goes out of scope. If there is no copy semantic required, it can be used in containers and algorithms of the Standard Template Library. `std::unique_ptr` is as cheap and fast as a raw pointer, if you use no special deleter.

> ⚠ **Don't use std::auto_ptr**
> Classical C++03 has a smart pointer `std::auto_ptr`, which exclusively takes care of the lifetime of a resource. But `std::auto_ptr` has a conceptional issue. If you implicitly or explicitly copy an `std::auto_ptr`, the resource is moved. So instead of the copy semantic, you have a hidden move semantic, and therefore you often have undefined behavior. So `std::auto_ptr` is *deprecated* in C++11 and you should use instead `std::unique_ptr`. You can neither implicitly or explicitly copy an `std::unique_ptr`. You can only move it.

```
#include <iostream>
#include <memory>
int main(){
  std::auto_ptr<int> ap1(new int(2011));
  std::auto_ptr<int> ap2 = ap1;              // OK

  std::unique_ptr<int> up1(new int(2011));
  //std::unique_ptr<int> up2 = up1;            // ERROR
  std::unique_ptr<int> up3 = std::move(up1); // OK
```

```
  return 0;
}
```

Runtime Error

These are the methods of `std::unique_ptr`:

Name	Description
`get`	Returns a pointer to the resource.
`get_deleter`	Returns the delete function.
`release`	Returns a pointer to the resource and releases it.
`reset`	Resets the resource.
`swap`	Swaps the resources.

Methods of `std::unique_ptr`

In the following code sample you can see the application of these methods:

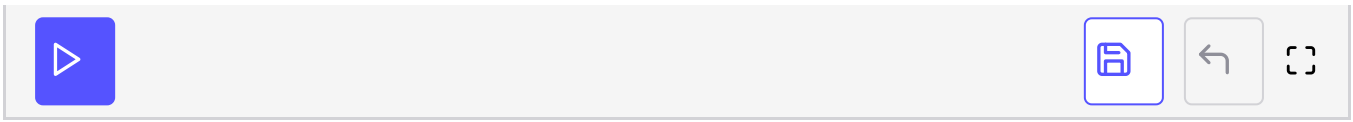
```
// uniquePtr.cpp
#include <iostream>
#include <utility>
#include <memory>

using namespace std;

struct MyInt{
  MyInt(int i):i_(i){}
  ~MyInt(){
    cout << "Good bye from " << i_ << endl;
  }
  int i_;
};

int main(){
```

```
  unique_ptr<MyInt> uniquePtr1{new MyInt(1998)};
  cout << uniquePtr1.get() << endl;               // 0x15b5010


  unique_ptr<MyInt> uniquePtr2{move(uniquePtr1)};
  cout << uniquePtr1.get() << endl;               // 0
  cout << uniquePtr2.get() << endl;               // 0x15b5010
  {
    unique_ptr<MyInt> localPtr{new MyInt(2003)};
  }                                               // Good bye from 2003
  uniquePtr2.reset(new MyInt(2011));              // Good bye from 1998
  MyInt* myInt= uniquePtr2.release();
  delete myInt;                                   // Good by from 2011

  unique_ptr<MyInt> uniquePtr3{new MyInt(2017)};
  unique_ptr<MyInt> uniquePtr4{new MyInt(2022)};
  cout << uniquePtr3.get() << endl;               // 0x15b5030
  cout << uniquePtr4.get() << endl;               // 0x15b5010

  swap(uniquePtr3, uniquePtr4);
  cout << uniquePtr3.get() << endl;               // 0x15b5010
  cout << uniquePtr4.get() << endl;               // 0x15b5030
  return 0;
}
```

`std::unique_ptr` has a specialisation for arrays:

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

using namespace std;

class MyStruct{
public:
  MyStruct():val(count){
    cout << static_cast<void*>(this) << " Hello: " << val << endl;
    MyStruct::count++;
  }
  ~MyStruct(){
    cout << static_cast<void*>(this) << " Good Bye: " << val << endl;
    MyStruct::count--;
  }
private:
  int val;
  static int count;
};

int MyStruct::count= 0;

int main(){
  {
    // generates a myUniqueArray with thre `MyStructs`
    unique_ptr<MyStruct[]> myUniqueArray{new MyStruct[3]};
  }
  // 0x1200018 Hello: 0
```

```
  // 0x120001c Hello: 1
  // 0x1200020 Hello: 2
  // 0x1200020 GoodBye: 2

  // 0x120001c GoodBye: 1
  // 0x1200018 GoodBye: 0

  return 0;
}
```



`std::unique_ptr` array

Special Deleters

`std::unique_ptr` can be parametrized with special deleters: `std::unique_ptr<int, MyIntDeleter> up(new int(2011), myIntDeleter())`. `std::unique_ptr` uses by default the deleter of the resource.

std::make_unique

The helper function std::make_unique was unlike its sibling std::make_shared forgotten in the C++11 standard. So `std::make_unique` was added with the C++14 standard. `std::make_unique` enables it to create a `std::unique_ptr` in a single step: `std::unique_ptr<int> up = std::make_unique<int>(2014)`.

In the next lesson, we will discuss shared pointers.