- Examples

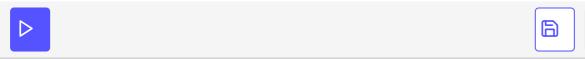
The key question of the std::unique_ptr is when to delete the underlying resource. This occurs when the std::unique_ptr goes out of scope or receives a new resource. Let's look at two use cases to better understand this concept.

WE'LL COVER THE FOLLOWING ^ Example 1 Explanation Example 2 Explanation

Example 1

```
// uniquePtr.cpp
                                                                                            G
#include <iostream>
#include <memory>
#include <utility>
struct MyInt{
 MyInt(int i):i_(i){}
  ~MyInt(){
    std::cout << "Good bye from " << i_ << std::endl;
  int i_;
};
int main(){
  std::cout << std::endl;</pre>
  std::unique_ptr<MyInt> uniquePtr1{ new MyInt(1998) };
  std::cout << "uniquePtr1.get(): " << uniquePtr1.get() << std::endl;</pre>
  std::unique_ptr<MyInt> uniquePtr2;
```

```
uniquertrz= Sta::move(uniquertri);
std::cout << "uniquePtr1.get(): " << uniquePtr1.get() << std::endl;</pre>
std::cout << "uniquePtr2.get(): " << uniquePtr2.get() << std::endl;</pre>
std::cout << std::endl;</pre>
  std::unique_ptr<MyInt> localPtr{ new MyInt(2003) };
std::cout << std::endl;</pre>
uniquePtr2.reset(new MyInt(2011));
MyInt* myInt= uniquePtr2.release();
delete myInt;
std::cout << std::endl;</pre>
std::unique_ptr<MyInt> uniquePtr3{ new MyInt(2017) };
std::unique_ptr<MyInt> uniquePtr4{ new MyInt(2022) };
std::cout << "uniquePtr3.get(): " << uniquePtr3.get() << std::endl;</pre>
std::cout << "uniquePtr4.get(): " << uniquePtr4.get() << std::endl;</pre>
std::swap(uniquePtr3, uniquePtr4);
std::cout << "uniquePtr3.get(): " << uniquePtr3.get() << std::endl;</pre>
std::cout << "uniquePtr4.get(): " << uniquePtr4.get() << std::endl;</pre>
std::cout << std::endl;</pre>
```



Explanation

- The class MyInt (line 7 -17) is a simple wrapper for a number. We have adjusted the destructor in line 11 - 13 for observing the life cycle of MyInt.
- We create, in line 24, a std::unique_ptr and return, in line 26, the
 address of its resource new MyInt(1998). Afterward, we move the
 uniquePtr1 to uniquePtr2 (line 29). Therefore, uniquePtr2 is the owner of
 the resource. That is shown in the output of the program in lines 30 and
 31.
- In line 37, the local std::unique_ptr reaches its valid range at the end of
 the scope. Therefore, the destructor of the localPtr meaning the
 destructor of the resource new MyInt(2003) will be executed.

- The most interesting lines are lines 42 to 44. At first, we assign a new resource to we assign the uniquePtr1. Therefore, the destructor of MyInt(1998) will be executed. After the resource in line 43 is released, we can explicitly invoke the destructor.
- The rest of the program is quite easy to understand. In lines 48 58, we create two std::unique_ptr and swap their resources. std::swap uses move semantic since std::unique_ptr supports no copy semantic. With the end of the main function, uniquePtr3 and uniquePtr4 go out of scope, and their destructor will automatically be executed.

Now that we have a sense of this technique, let's dig into a few details of std::unique_ptr in the example below.

Example 2

std::unique_ptr has a specialization for arrays. The access is transparent,
meaning that if the std::unique_ptr manages the lifetime of an object, the
operators for the object access are overloaded (operator* and operator->). If
std::unique_ptr manages the lifetime of an array, the index operator [] is
overloaded. The invocations of the operators are, therefore, transparently
forwarded to the underlying resource.

```
// uniquePtrArray.cpp

#include <iomanip>
#include <iostream>
#include <memory>

class MyStruct{
public:
    MyStruct(){
        std::cout << std::setw(15) << std::left << (void*) this << " Hello " << std::endl;
}
        ~MyStruct(){
        std::cout << std::setw(15) << std::left << (void*)this << " Good Bye " << std::endl;
};

int main(){
    std::cout << std::endl;

std::cout << std::endl;

std::cout << "**uniqInt(new int(2011));

std::cout << "**uniqInt (new int(2011));

std::cout << "**uniqInt (new int(2011));
</pre>
```

```
std::cout << std::endl;

{
    std::unique_ptr<MyStruct[]> myUniqueArray{new MyStruct[5]};
}

std::cout << std::endl;

{
    std::unique_ptr<MyStruct[]> myUniqueArray{new MyStruct[1]};
    MyStruct myStruct;
    myUniqueArray[0]=myStruct;
}

std::cout << std::endl;

{
    std::unique_ptr<MyStruct[]> myUniqueArray{new MyStruct[1]};
    MyStruct myStruct;
    myStruct = myUniqueArray[0];
}

std::cout << std::endl;
}

std::cout << std::endl;
}</pre>
```







ני

Explanation

- We dereference (line 22) a std::unique_ptr and get the value of its resource.
- In lines 7 15, MyStruct acts as the base of an array of std::unique_ptr's. If we instantiate a MyStruct object, we will get its address. The destructor gives the output. Now it is easy to observe the life cycle of the objects.
- In lines 26 28, we create and destroy five instances of MyStruct.
- The lines 32 36 are more interesting. We create a MyStruct instance on the heap (line 33) and on the stack (line 34). Therefore, both objects have addresses from different ranges.
- Afterward, we assign the local object to the std::unique_pr (line 35). The lines 40 44 follows a similar strategy. Now we assign, the local object, the first element of myUniqueArray. The index access to the std::unique_ptr in the lines 35 and 43 feels like familiar to index access to an array.

Let's move on to the second type of smart pointers in this section, called shared pointers.