Smart Pointers: Passing Smart Pointers

In this lesson, we will discuss the rules regarding passing smart pointers.

WE'LL COVER THE FOLLOWING ^

- The Six Rules
 - R.32
 - R.33
 - R.34, R.35, and R.36
 - R.37

Passing smart pointers is an important topic that is seldom addressed. This process ends with the C++ core guidelines since they have six rules for passing std::shared_ptr and std::unique_ptr.

The Six Rules

The following six rules violate the import dry (don't repeat yourself) principle for software development. At the end, we have only four rules, which makes life as a software developer a lot easier. Here are the rules.

- 1. R.32: Take a unique_ptr<widget> parameter to express that a function
 assumes ownership of a widget
- 2. **R.33**: Take a unique_ptr<widget>& parameter to express that a function reseats the widget
- 3. R.34: Take a shared_ptr<widget> parameter to express that a function is part owner
- 4. R.35: Take a shared_ptr<widget>& parameter to express that a function
 might reseat the shared pointer
- 5. **R.36**: Take a const <a href="mailto:shared_ptr<widget>&">shared_ptr<widget>&">shared_ptr<widget>&">shared_ptr<widget>&">parameter to express that it might retain a reference count to the object ???

6. **R.37**: Do not pass a pointer or reference obtained from an aliased smart pointer

Let's start with the first two rules for std::unique_ptr.

R.32

If a function should take ownership of a Widget, you should take the std::unique_ptr<Widget> by copy. The consequence is that the caller has to move the std::unique ptr<Widget> to make the code run.

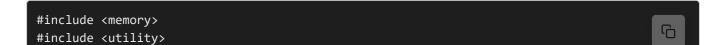
```
#include <memory>
                                                                                         G
#include <utility>
struct Widget{
   Widget(int){}
};
void sink(std::unique_ptr<Widget> uniqPtr){
   // do something with uniqPtr
}
int main(){
   auto uniqPtr = std::make_unique<Widget>(1998);
   sink(std::move(uniqPtr));
                                   // (1)
    sink(uniqPtr);
                                   // (2) ERROR
```

The call in line 15 is fine but the call line 16 breaks because you cannot copy an std::unique_ptr. If your function only wants to use the Widget, it should take its parameter by pointer or by reference. A pointer can be a null pointer, but a reference cannot.

```
void useWidget(Widget* wid);
void useWidget(Widget& wid);
```

R.33

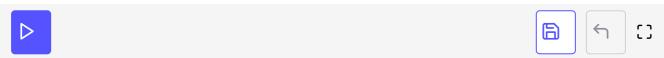
Sometimes a function wants to reset a <code>Widget</code>. In this use-case, you should pass the <code>std::unique_ptr<Widget></code> by a non-const reference.



```
struct Widget{
    Widget(int){}
};

void reseat(std::unique_ptr<Widget>& uniqPtr){
    uniqPtr.reset(new Widget(2003)); // (0)
    // do something with uniqPtr
}

int main(){
    auto uniqPtr = std::make_unique<Widget>(1998);
    reseat(std::move(uniqPtr)); // (1) ERROR
    reseat(uniqPtr); // (2)
}
```



Now, the call in line 16 fails because we cannot bind an rvalue to a non-const lvalue reference. This will not hold for the copy in line 17. An lvalue can be bound to an lvalue reference. The call in line 9 will not only construct a new Widget(2003), but it will also destruct the old Widget(1998).

The next three rules of std::shared_ptr are repetitions, so we will only discuss one.

R.34, R.35, and R.36

Here are the three function signatures that we have to address.

```
void share(std::shared_ptr<Widget> shaWid);
void reseat(std::shard_ptr<Widget>& shadWid);
void mayShare(const std::shared_ptr<Widget>& shaWid);
```

Let's take a look at each function signature in isolation, but what does this mean from the function perspective? Let's find out!

- void share(std::shared_ptr shaWid): I'm for the lifetime of the function body a shared owner of the Widget. At the start of the function body, I will increase the reference counter; at the end of the function, we will decrease the reference counter; therefore, the Widget will stay alive, as long as I use it.
- void reseat(std::shared_ptr& shaWid): I'm not a shared owner of the

Widget because I will not change the reference counter. I have not guaranteed that the Widget will stay alive during the execution of my function, but I can reseat the resource. A non-const lvalue reference is more like: I borrow the resource and can reseat it.

• void mayShare(const std::shared_ptr& shaWid): I only borrow the resource. Neither can I extend the lifetime of the resource nor can I reseat the resource. To be honest, you should use a pointer (Widget*) or a reference (Widget&) as a parameter instead, because there is no added value in using an std::shared_ptr.

R.37

Let's take a look at a short code snippet to makes the rule clearer.

globShared in line 13 is a globally shared pointer. The function shared takes its argument per reference in line 5. Therefore, the reference counter of shaPtr will not be increased and the function share will not extend the lifetime of Widget(2011). The issue begins on line 7. oldFunc accepts a pointer to the Widget; therefore, oldFunc has no guarantee that the Widget will stay alive during its execution. oldFunc only borrows the Widget.

The solution is quite simple. We must ensure that the reference count of <code>globShared</code> will be increased before the call to the function <code>oldFunc</code>, meaning that we must make a copy of <code>std::shared_ptr</code>:

Dogs the attended by converte the function change

• Pass the sta::snared_ptr by copy to the function shared.

```
void shared(std::shared_ptr<Widget> shaPtr){
  oldFunc(*shaPtr);
  // do something with shaPtr
}
```

• Make a copy of the **shaPtr** in the function shared:

```
void shared(std::shared_ptr<Widget>& shaPtr){
   auto keepAlive = shaPtr;
   oldFunc(*shaPtr);
   // do something with keepAlive or shaPtr
}
```

The same reasoning also applies to std::unique_ptr, but there isn't a simple
solution since we cannot copy an std::unique_ptr. Rather, we can clone the
std::unique_ptr and make a new std::unique_ptr.

Now that we have gone over **Reduced Resources** in Embedded Programming with Modern C++, we will discuss how to implement **Several Tasks Simultaneously** in embedded programming in the next chapter.