In the previous section, we have taken a look at the three smart pointer types in C++. In addition to smart pointers, you are now also familiar with move semantics, which is of particular importance in this section. In the following, we will discuss how to properly pass and return smart pointers to functions and vice-versa. In modern C++, there are various ways of doing this and in many cases, the method of choice has an impact on both performance and code robustness. The basis of this section are the C++ core guidelines on smart pointers, some of which we will be examining in the following.

**Passing smart pointers to functions**

Let us consider the following recommendation of the C++ guidelines on smart pointers:

**R. 30 : Take smart pointers as parameters only to explicitly express lifetime semantics**

The core idea behind this rule is the notion that functions that only manipulate objects without affecting its lifetime in any way should not be concerned with a particular kind of smart pointer. A function that does not manipulate the lifetime or ownership should use raw pointers or references instead. A function should take smart pointers as parameter only if it examines or manipulates the smart pointer itself. As we have seen, smart pointers are classes that provide several features such as counting the references of a shared\_ptr or increasing them by making a copy. Also, data can be moved from one unique\_ptr to another and thus transferring the ownership. A particular function should accept smart pointers only if it expects to do something of this sort. If a function just needs to operate on the underlying object without the need of using any smart pointer property, it should accept the objects via raw pointers or references instead.

The following examples are **pass-by-value types that lend the ownership** of the underlying object:

1. void f(std::unique\_ptr<MyObject> ptr)
2. void f(std::shared\_ptr<MyObject> ptr)
3. void f(std::weak\_ptr<MyObject> ptr)

Passing smart pointers by value means to lend their ownership to a particular function f. In the above examples 1-3, all pointers are passed by value, i.e. the function f has a private copy of it which it can (and should) modify. Depending on the type of smart pointer, a tailored strategy needs to be used. Before going into details, let us take a look at the underlying rule from the C++ guidelines (where "widget" can be understood as "class").

## Unique Pointer Passing

The basic idea of a unique\_ptr is that there exists only a single instance of it. This is why it can’t be copied to a local function but needs to be moved instead with the function std::move. The code example on the right illustrates the principle of transferring the object managed by the unique pointer uniquePtr into a function f.

The class MyClass has a private object \_member and a public function printVal() which prints the address of the managed object (this) as well as the member value to the console. In main, an instance of MyClass is created by the factory function make\_unique() and assigned to a unique pointer instance uniquePtr for management. Then, the pointer instance is moved into the function f using move semantics. As we have not overloaded the move constructor or move assignment operator in MyClass, the compiler is using the default implementation. In f, the address of the copied / moved unique pointer ptr is printed and the function printVal() is called on it. When the path of execution returns to main(), the program checks for the validity of uniquePtr and, if valid, calls the function printVal() on it again. Here is the console output of the program:

unique\_ptr 0x7ffeefbff710, managed object 0x100300060 with val = 23

unique\_ptr 0x7ffeefbff6f0, managed object 0x100300060 with val = 23

The output nicely illustrates the copy / move operation. Note that the address of unique\_ptr differs between the two calls while the address of the managed object as well as of the value are identical. This is consistent with the inner workings of the move constructor, which we overloaded in a previous section. The copy-by-value behavior of f() creates a new instance of the unique pointer but then switches the address of the managed MyClass instance from source to destination. After the move is complete, we can still use the variable uniquePtr in main but it now is only an empty shell which does not contain an object to manage.

## Unique Pointer Passing

#include <iostream>

#include <memory>

class MyClass

{

private:

int \_member;

public:

MyClass(int val) : \_member{val} {}

void printVal() { std::cout << ", managed object " << this << " with val = " << \_member << std::endl; }

};

void f(std::unique\_ptr<MyClass> ptr)

{

std::cout << "unique\_ptr " << &ptr;

ptr->printVal();

}

int main()

{

std::unique\_ptr<MyClass> uniquePtr = std::make\_unique<MyClass>(23);

std::cout << "unique\_ptr " << &uniquePtr;

uniquePtr->printVal();

f(std::move(uniquePtr));

if (uniquePtr)

uniquePtr->printVal();

return 0;

}

## Shared Pointer Passing

When passing a shared pointer by value, move semantics are not needed. As with unique pointers, there is an underlying rule for transferring the ownership of a shared pointer to a function:

**R.34: Take a shared\_ptr parameter to express that a function is part owner**

Consider the example on the right. The main difference in this example is that the MyClass instance is managed by a shared pointer. After creation in main(), the address of the pointer object as well as the current reference count are printed to the console. Then, sharedPtr is passed to the function f() by value, i.e. a copy is made. After returning to main, pointer address and reference counter are printed again. Here is what the output of the program looks like:

shared\_ptr (ref\_cnt= 1) 0x7ffeefbff708, managed object 0x100300208 with val = 23

shared\_ptr (ref\_cnt= 2) 0x7ffeefbff6e0, managed object 0x100300208 with val = 23

shared\_ptr (ref\_cnt= 1) 0x7ffeefbff708, managed object 0x100300208 with val = 23

Throughout the program, the address of the managed object does not change. When passed to f() , the reference count changes to 2. After the function returns and the local shared\_ptr is destroyed, the reference count changes back to 1. In summary, move semantics are usually not needed when using shared pointers. Shared pointers can be passed by value safely and the main thing to remember is that with each pass, the internal reference counter is increased while the managed object stays the same.

Without giving an example here, the weak\_ptr can be passed by value as well, just like the shared pointer. The only difference is that the pass does not increase the reference counter.

## Shared Pointer Passing

#include <iostream>

#include <memory>

void f(std::shared\_ptr<MyClass> ptr)

{

std::cout << "shared\_ptr (ref\_cnt= " << ptr.use\_count() << ") " << &ptr;

ptr->printVal();

}

int main()

{

std::shared\_ptr<MyClass> sharedPtr = std::make\_shared<MyClass>(23);

std::cout << "shared\_ptr (ref\_cnt= " << sharedPtr.use\_count() << ") " << &sharedPtr;

sharedPtr->printVal();

f(sharedPtr);

std::cout << "shared\_ptr (ref\_cnt= " << sharedPtr.use\_count() << ") " << &sharedPtr;

sharedPtr->printVal();

return 0;

}

With the above examples, pass-by-value has been used to lend the ownership of smart pointers. Now let us consider the following additional rules from the C++ guidelines on smart pointers:

**R.33: Take a unique\_ptr& parameter to express that a function reseats the widget**

and

**R.35: Take a shared\_ptr& parameter to express that a function might reseat the shared pointer**

Both rules recommend passing-by-reference, when the function is supposed to modify the ownership of an existing smart pointer and not a copy. We pass a non-const reference to a unique\_ptr to a function if it might modify it in any way, including deletion and reassignment to a different resource.

Passing a unique\_ptr as const is not useful as the function will not be able to do anything with it: Unique pointers are all about proprietary ownership and as soon as the pointer is passed, the function will assume ownership. But without the right to modify the pointer, the options are very limited.

A shared\_ptr can either be passed as const or non-const reference. The const should be used when you want to express that the function will only read from the pointer or it might create a local copy and share ownership.

Lastly, we will take a look at **passing raw pointers** and references. The general rule of thumb is that we can use a simple raw pointer (which can be null) or a plain reference (which can  be null), when the function we are passing will only inspect the managed object without modifying the smart pointer. The internal (raw) pointer to the object can be retrieved using the get() member function. Also, by providing access to the raw pointer, you can use the smart pointer to manage memory in your own code and pass the raw pointer to code that does not support smart pointers.

When using raw pointers retrieved from the get() function, you should take special care not to delete them or to create new smart pointers from them. If you did so, the ownership rules applying to the resource would be severely violated. When passing a raw pointer to a function or when returning it (see next section), raw pointers should always be considered as owned by the smart pointer from which the raw reference to the resource has been obtained.

## Returning smart pointers from functions

With return values, the same logic that we have used for passing smart pointers to functions applies: Return a smart pointer, both unique or shared, if the caller needs to manipulate or access the pointer properties. In case the caller just needs the underlying object, a raw pointer should be returned.

Smart pointers should always be returned by value. This is not only simpler but also has the following advantages:

1. The overhead usually associated with return-by-value due to the expensive copying process is significantly mitigated by the built-in move semantics of smart pointers. They only hold a reference to the managed object, which is quickly switched from destination to source during the move process.
2. Since C++17, the compiler used Return Value Optimization (RVO) to avoid the copy usually associated with return-by-value. This technique, together with copy-elision, is able to optimize even move semantics and smart pointers (not in call cases though, they are still an essential part of modern C++).
3. When returning a shared\_ptr by value, the internal reference counter is guaranteed to be properly incremented. This is not the case when returning by pointer or by reference

The topic of smart pointers is a complex one. In this course, we have covered many basics and

some of the more advanced concepts. However, there are many more aspects to consider and features to use when integrating smart pointers into your code. The full set of smart pointer rules in the C++ guidelines is a good start to dig deeper into one of the most powerful features of modern C++.

## Best-Practices for Passing Smart Pointers

This sections contains a condensed summary of when (and when not) to use smart pointers and how to properly pass them between functions. This section is intended as a guide for your future use of this important feature in modern C++ and will hopefully encourage you not to ditch raw pointers altogether but instead to think about where your code could benefit from smart pointers - and when it would most probably not.

The following list contains all the variations (omitting const) of passing an object to a function:

**void** **f**( object\* ); *// (a)*

**void** **f**( object& ); *// (b)*

**void** **f**( unique\_ptr<object> ); *// (c)*

**void** **f**( unique\_ptr<object>& ); *// (d)*

**void** **f**( shared\_ptr<object> ); *// (e)*

**void** **f**( shared\_ptr<object>& ); *// (f)*

### The Preferred Way

The preferred way of to pass object parameters is by using a) or b) :

In doing so, we do not have to worry about the lifetime policy a caller might have implemented. Using a specific smart pointer in a case where we only want to observe an object or manipulate a member might be overly restrictive

With the non-owning raw pointer \* or the reference & we can observe an object from which we can assume that its lifetime will exceed the lifetime of the function parameter. In concurrency however, this might not be the case, but for linear code we can safely assume this.

To decide wether a \* or & is more appropriate, you should think about wether you need to express that there is no object. This can only be done with pointers by passing e.g. nullptr. In most other cases, you should use a reference instead.

### The Object Sink

The preferred way of passing an object to a function so that the function takes ownership of the object (or „consumes“ it) is by using method c) from the above list:

void f( unique\_ptr<object> );

In this case, we are passing a unique pointer by value from caller to function, which then takes ownership of the the pointer and the underlying object. This is only possible using move semantics as there may be only a single reference to the object managed by the unique pointer.

After the object has been passed in this way, the caller will have an invalid unique pointer and the function to which the object now belongs may destroy it or move it somewhere else.

Using const with this particular call does not make sense as it models an ownership transfer so the source will be definitely modified.

### In And Out Again 1

In some cases, we want to modify a unique pointer (not necessarily the underlying object) and re-use it in the context of the caller. In this case, method d) from the above list might be most suitable:

void f( unique\_ptr<object>& );

Using this call structure, the function states that it might modify the smart pointer, e.g. by redirecting it to another object. It is not recommended to use it for accepting an object only because we should avoid restricting ourselves unnecessarily to a particular object lifetime strategy on the caller side.

Using const with this call structure is not recommendable as we would not be able to modify the unique\_ptr in this case. In case you want to modify the underlying object, use method a) instead.

### Sharing Object Ownership

In the last examples, we have looked at strategies involving unique ownership. In this example, we want to express that a function will store and share ownership of an object on the heap. This can be achieved by using method e) from the list above:

void f( shared\_ptr<object> )

In this example, we are making a copy of the shared pointer passed to the function. In doing so, the internal reference counter within all shared pointers referring to the same heap object is incremented by one.

This strategy can be recommended for cases where the function needs to retain a copy of the shared\_ptr and thus share ownership of the object. This is of interest when we need access to smart pointer functions such as the reference count or we must make sure that the object to which the shared pointer refers is not prematurely deallocated (which might happen in concurrent programming).

If the local scope of the function is not the final destination, a shared pointer can also be moved, which does not increase the reference count and is thus more effective.

A disadvantage of using a shared\_ptr as a function argument is that the function will be limited to using only objects that are managed by shared pointers - which limits flexibility and reusability of the code.

### In And Out Again 2

As with unique pointers, the need to modify shared pointers and re-use them in the context of the caller might arise. In this case, method f) might be the right choice:

void f( shared\_ptr<object>& );

This particular way of passing a shared pointer expresses that the function f will modify the pointer itself. As with method e), we will be limiting the usability of the function to cases where the object is managed by a shared\_ptr and nothing else.

## Last Words

The topic of smart pointers is a complex one. In this course, we have covered many basics and some of the more advanced concepts. However, for some cases there are more aspects to consider and features to use when integrating smart pointers into your code. The [**full set of smart pointer rules**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#rsmart-smart-pointers) in the C++ guidelines is a good start to dig deeper into one of the most powerful features of modern C++.