Page 1;

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").

PAge 2: video

Page 3:

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

```
#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;
}
```

Page 4:

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

#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;
}
```

Page5:

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 ~~not~~ 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

Page 6:

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++.