**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) :

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

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

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++.

**Outro**