C++ Smart Pointers Ref Card - join cppstories.com to get latest updates @this link

Common

- Smart pointers are located in the `<memory>` header.
- Smart pointers are "smart" because they hold a pointer to an object/resource plus they know the ownership of the pointer. The are based on the RAII pattern.
- `unique_ptr` and `shared_ptr` have overloaded access operators `*` and `->`, so smart pointers can be dereferenced like regular raw pointers.
- Use `.get()` (on `unique_ptr` and `shared_ptr`) to access the raw, underlying pointer.
- `.get()` is useful when you want to pass a pointer to a function to "observe" the managed object
```
void useObject(MyType* pObj) { }
useObject(mySmartPtr.get());
```
- `unique_ptr` (since C++11) and `shared_ptr` (since C++17) have template specialization for arrays (`delete[]` will be called on clean up). This might be helpful when you get a pointer to an array from some third-party library or a legacy system. Still, if possible, it's better to use some standard containers like `std::vector` or `std::array`.
- **Reminder**: don't use `auto_ptr`! It has been deprecated since C++11 and removed in C++17. Replace it with `unique_ptr`.
- You can try with `modernize-replace-auto-ptr` from Clang Tidy to automate refactoring.
- In C++17/C++20, there is no class template argument deduction (CTAD) for smart pointers. It is impossible for the compiler to distinguish a pointer obtained from an array and non-array forms of `new()`.
- Since C++20 there are atomic smart pointers `std::atomic<std::shared_ptr<T>>` and `std::atomic<std::weak_ptr<T>>`. C++20 also deprecates global atomic functions for smart pointers available since C++11.
- C++20 adds various `*_for_overwrite` creation functions which take no constructor arguments and use default-initialization (equivalent to `new T`). This avoids unnecessary initialization in situations where the initial value is never read (like reading into a buffer)

References

cppreference.com/cpp/memory
CppCoreGuidelines/Resource
cppstories.com
C++20 changes - P2131

Get Updates to this ref card @this link

std::unique_ptr

A lightweight smart pointer that has the unique ownership of a managed object.

- Unique pointer destroys the underlying object when it goes out of scope, its `reset()` method is called or is assigned with a new pointer/object.
- `unique_ptr` is movable, but not copyable.
- Usually, it's the size of a single native pointer (for stateless deleters, or two pointers when a pointer for deleter is required).

Creation

Advised with `auto` and `std::make_unique`:
```
auto pObj = make_unique<MyType>(…)'
```
or with explicit new:
```
unique_ptr<MyType> pObject(new MyType(…))
```
but the type occurs twice here, and you need to use raw new which is not considered a modern approach.

Custom deleters

A deleter is a callable object used to delete a resource. By default it uses `delete` or `delete[]`. Type of the deleter is part of the type of the `unique_ptr`.
```
struct DelFn {
    void operator()(MyTy* p) {
        p->SpecialDelete();
        delete p;
    }
};
using my_ptr = unique_ptr<MyTy, DelFn>;
```

- Deleter is not called when pointer is null
- `get_deleter()` can return a non const reference to the deleter, so it can be used to replace it

Passing to functions

`unique_ptr` is movable only, so it should be passed with `std::move` to explicitly express the ownership transfer:
```
auto pObj = make_unique<MyType>(…);
func(std::move(pObj));
// pObj is invalid after the call!
```

Other

- `reset()` – resets the pointer (deletes the old one)
- `unique_ptr` is also useful in "pimpl" idiom implementation
- `unique_ptr` is usually the first candidate to return from factory functions. If factories gets more complicated (like when adding caches), you might then use `shared_ptr` (or `weak_ptr`)

std::shared_ptr

Multiple shared pointers can point to the same object, sharing the ownership. When the last shared pointer goes out of scope, the managed object is deleted. This technique is implemented through reference counting.

- `shared_ptr` is copyable and movable
- it's usually the size of two native pointers: one for the object and one to point at the control block.
- The control block usually holds the reference counter, weak counter, deleter and allocator.

Creation

Advised method is through `std::make_shared()`:
```
auto pObj = make_shared<MyType>(…)
```
`make_shared` will usually allocate the control block next to the Object, so there's better memory locality.

Custom deleters

A deleter is stored in a control block and can be passed during creation (not with `make_shared()`). Deleter is not part of the type and can be anything callable.
```
void DelFn(MyTp* p) {
    if (p) p->OnDelete();
    delete p;
}
shared_ptr<MyTp> ptr(new MyTp(), DelFn);
```

- A deleter must cope with null pointer values. A Deleter might be called when the pointer is empty.
- `get_deleter()` (non-member function) returns a non const pointer to the deleter

Passing to functions

To share the ownership pass a shared pointer by value. Reference counter is updated atomically, so you need to be aware of the extra synchronisation cost. `std::move` can be also used to transfer the ownership.
To observe the object use `.get()`.

Other

- The reference counter access is atomic but the pointer access is not thread-safe.
- Use `shared_from_this()` to return a shared pointer to `*this`. The class must derive from `std::enable_shared_from_this`.
- Casting between pointer types can be done using `dynamic_pointer_cast`, `static_pointer_cast` or `reinterpret_pointer_cast`.
- `shared_ptr` might create cyclic dependencies and mem leaks when two pointers point to each other.

std::weak_ptr

Non-owning smart pointer that holds a "weak" reference to an object that is managed by `std::shared_ptr`. It must be converted to `std::shared_ptr` to access the referenced object – via the `lock()` method.

- One example where weak pointers are useful is caching. Such system distributes only weak pointers, and before any use, the client is responsible for checking if the resource is still alive.
- A weak pointer is also used to break cycles in shared pointers.

Creation

A weak pointer is created from a `shared_ptr`, but before using it, you have to convert it to shared_ptr again.
```
weak_ptr pWeak = pSharedPtr;
if (auto observe = pWeak.lock()) {
    // the object is alive
} else {
   // shared_ptr was deleted
}
```

- A weak pointer created from a shared pointer will increase 'weak reference counter' that is stored in the control block of the shared pointer. Even if all shared pointers (referring to a single object) are dead, but one weak pointer has a weak reference (to that object) the control block might still be present in the memory. This might be a problem when the control block is allocated together with the object (like when using `make_shared`). In that case, the destructor of the object is called, but memory is not released.

Other

- `use_count()` – returns the number of shared pointers sharing the same managed object.
- Use `expired()` to check if the managed object is still present.
- The weak pointer doesn't have `*` and `->` operators overloaded, so you cannot dereference underlying pointer before converting to `shared_ptr` (via `lock()`).

See more about modern C++

C++ STORIES

© 2021 Bartlomiej Filipek, cppstories.com