C++11 BLWS (Wednesday 1)

Smart Pointers

Short breaks will be inserted as convenient.

C++11: `std::shared_ptr`

With `std::shared_ptr` C++11 introduced a Smart Pointer type that does reference counting on its pointee:

- When created by its default constructor it points to no object and creates an owner (reference) count set to zero.
- When created and initialized with a bare pointer it assumes it is the first and only one and creates an owner count set to 1.
- When initialized from another of its kind it points to the same object as the other (if any) and increments the – then shared – owner count.
- When assigned from another of its kind it first decrements the owner count and if that drops to zero, destroys the pointee; then it continues as if it were initialized from the pointer assigned to it.
- When it goes out of scope it also decrements the owner count and if that drops to zero, destroys the pointee.

Shared Pointee Construction

Per default an `std::shared_ptr` is initialized with no pointee.

Given

```
class MyClass { … MyClass(bool, double, std::string); … };
```

an `std::shared_ptr` can be initialized to point to a heap allocated object of type `MyClass` as follows:[*]

```
std::shared_ptr<MyClass> p1{new MyClass(true, 3.14, "hi!")};
```

Or:

```
auto p2 = std::make_shared<MyClass>(true, 3.14, "hi!");
```

The usual recommendation is to prefer the second way over the first as it can reserve space for `MyClass` together with the helper object (holding owner and observer count[*]) in a single heap allocation.

---

[*]: The observer count has not yet been introduced; its purpose will become clear when `std::weak_ptr` gets explained.

Shared Pointee Access

Given an `std::shared_ptr<SomeType> p` the most typical access to the pointee is via overloaded `operator*` or `operator->`, possibly after a testing whether there is an object:*

```
if (p) … *p …; // access whole object
       … p->m …; // access data member m
       … p->f(); // call member function f
```

Furthermore `p.get()` returns the address of the pointee (or `nullptr`), so

- it can bridge between `std::shared_ptr` and legacy code that expects a native pointer,
- at least as long as the recipient is short-lived (compared to `p`) and
- **does not assume ownership**.

Shared Pointee Sharing

Sharing the pointee is the *raison d'être* of a an `std::share_ptr`. Therefore it is clear that the following code

```
auto p1 = std::make_shared<T>( … ); // create T with c'tor arguments
…                                 // ^^^--- as supplied here in parentheses
auto p2 = p1;          // OK
```

will create pointee which is then shared between `p1` and `p2` but with

```
auto p3 = p1.get()
```

`p3` assumes to be the sole owner of the object created as pointee for `p1`.

> **[!]** According to the standard this is *Undefined Behavior* which in practice will typically lead dangling pointers …*

*: … and depending on the whether and when the released memory is put into use again to unpredictable aftereffects.

Shared From `this`

The same problem arises slightly disguised if an object already managed by a shared pointer needs "to return itself":*

```
class MyClass {
    // ...
    MyClass* myself() { return this; }
};
```

[!] If the address returned by `MyClass::myself` is used to initialize an `std::shared_ptr<MySelf>` the problem is **exactly** the one discussed on the previous page.

This is the recipe to follow:

```
class MyClass : public std::enable_shared_from_this {
    // ...
 std::shared_ptr<MyClass> myself() { return shared_from_this(); }
};
```

*: A closeley related scenario is given when an object needs to register a member function bound to itself as call-back to be invoked later.

Shared Pointee Destruction

The default way to destruct the pointee (when the owner count drops to zero) is with `delete`.

If this is not appropriate a custom deleter can be specified at construction time:

```
std::shared_ptr<std::FILE> auto_close_fp{
    std::fopen("somefile", "r"),
    [](FILE *fp) { if (fp) std::fclose(fp); }
};
```

If the pointee is guaranteed to be valid or a custom deleter with a single pointer argument of pointee-type that is `nullptr`-safe* it could be specified directly:

```
if (auto fp = fopen("myfile", "w")) {
   std::shared_ptr<std::FILE> auto_close_fp{fp, std::fclose};
   …
}
```

*: Implementations of `fclose` differ in C, some are more robust and check the pointer argument for `(FILE*)0`, but according to the C-Standard an invalid pointer causes undefined behavior.

C++11: `std::weak_ptr`

With `std::weak_ptr` C++ introduced a companion to `std::shared_ptr`, mainly used to break cyclic references, which would otherwise defeat one of the main motivations for using smart pointers as a light-weight, high-efficiency garbage collector.

- An `std::weak_ptr` acts as **observer** of a pointee **owned** by an `std::shared_ptr`.
- As such it shares and manages an observer count (similar to but different from the owner count).
- A non-zero observer count **will not keep the pointee alive** if the owner count drops to zero.
- Therefore an `std::weak_ptr` has no way to access the pointee directly via overloaded `operator*` or `operator->`.
- To gain access it **first has to obtain an `std::shared_ptr`** – which might fail but if successful will keep the pointee alive even if all other owners cease to exist.

Weak Pointee Usage

Per default an `std::weak_ptr` is initialized with no pointee.

Given `std::shared_ptr<MyClass> p` an `std::weak_ptr` can be initialized to the pointee referred by `p` (if any) with:

```
std::weak_ptr<MyClass> wp{p};
```

To get access to the pointee an `std::shared_ptr` must be obtained and tested:

```
if (auto sp = wp.lock()) {
    // sp != nullptr
    // now owns the object wp had observed
    … *sp …;    // access whole object
    … sp.m …;  // access data member m
    … sp->f(); // call member function f
} // sp goes out of scope, if all other owners
  // are gone pointee will get destroyed here
```


C++11: `std::unique_ptr`

An `std::unique_ptr` is – as the name suggests – the sole owner of its pointee:

- Therefore there can always be only one for each pointee, i.e.
  - it is **not possible** to copy-construct or copy-assign an `std::unique_ptr` from another one of its kind (yields a compile time error), but
  - it **is possible** to move-construct or move-assign an `std::unique_ptr` from another one of its kind.
- When an `std::unique_ptr` goes out of scope or is re-assigned it first destroys its pointee (if any).

> The implementation of `std::unique_ptr` is very close to native pointers, i.e. same memory footprint, and also for most operations same performance, except those that need to care for destruction of a (previous) pointee.

Unique Pointee Construction

Per default an `std::unique_ptr` is initialized with no pointee.

Given any type `T` (a built-in type, a class from the standard library, or a user defined class) an `std::unique_ptr` can be initialized to point

- to a **single heap allocated object of this type***

```
std::unique_ptr<T> ptr{new T};
```

- or to an **array of `N` heap allocated objects**:

```
std::unique_ptr<T[]> arr{new T[N]};
```

> (i) The appropriate deleter is set depending on the template instantiation argument `T` or `T[]`.

*: Constructor arguments may be supplied as usual but there is no `std::make_unique` analogous to `std::make_shared` until C++14.

Unique Pointee Access

Given an `std::unique_ptr<SomeType> p` the most typical access to the pointee is via overloaded `operator*`, `operator->`, or `operator[]` if the pointee is an array, possibly after a testing whether there is an object:*

```
if (p) … *p …; // access whole object
       … p->m …; // access data member m
       … p->f(); // call member function f
       … p[i] …; // access i-th whole object (0-origin)
```

Furthermore

- `p.get()` returns the address of the pointee (or `nullptr`), so it can bridge between `std::unique_ptr` and legacy code that expects a native pointer with shorter lifetime as `p`;
- `p.release()` is similar but relinquishes ownership of the pointee **which the recipient has to assume then**.

Moving Unique Pointers

As unique pointers can not be copied, the (deliberate) use as initial values for copy constructor arguments is not possible:

```
extern std::unique_ptr<MyClass> make_MyClass();
…
std::unique_ptr<MyClass> p1 = make_MyClass(); // OK (move c'tor)
std::unique_ptr<MyClass> p2(p1); // ERROR (no copy c'tor exists)
```

Instead the exsting pointer needs to be moved:

```
std::unique_ptr<MyClass> p3(std::move(p1)) // OK (p1 nullptr now!)
```

Same for assignment – move works, copy does not:

```
p1 = p3;             // ERROR (no copy assignment exists)
p1 = make_MyClass(); // OK (move assignment)
p1 = std::move(p3);  // OK (p3 nullptr now!)
```

Unique Pointee Destruction

Per default the pointee is destructed with `delete` or `delete[]` depending on the way an `std::unique_ptr` has been created. Wrong pairing will not be detected at compile time but cause undefined runtime behaviour.*

```
{ std::unique_ptr<T[]> ptr{new T};
  …
} // destructor does delete[] on pointee address
```

Or:

```
{ std::unique_ptr<T> arr{new[N] T};
  …
} // destructor does delete on pointee address
```

> **[!]** Pairing plain allocation with array deallocation or array allocation with plain deallocation has undefined behavior.

*: In the best case this will cause an immediate crash with a good error message. But a crash may also occur much later with a misleading error message (if any) and therefore may be hard to relate to its original cause, or there may be a memory leak, memory overwritten with bad values, whatever …

Deprecated `std::auto_ptr`

The only Smart Pointer in C++98 was `std::auto_ptr`, which is deprecated since C++11.

- It had nearly the same behavior (and implementation) as `std::unique_ptr` has now, but C++98 had no means to forbid the copying versions of constructor and assignment while still allowing the move versions.

- Therefore `std::auto_ptr` had "copy"-constructor and -assignment which set their right hand side to `NULL`, i.e. the auto-pointer used for initialisation or from which the pointee was assigned lost its pointee.

- **That came as a surprise to some developers expecting a "more intelligent" behaviour from a "smart pointer".**

> Or as Bjarne Stroustrup once put it: With C++11 `std::unique_ptr` became what `std::auto_ptr` in C++98 always should have been but couldn't, due to lacking proper language support.

Boost: Smart Pointer

The term *Smart Pointer* is also used to subsume pointer-like helper classes in Boost:

- `boost::shared_ptr`:
  much like `std::shared_ptr` (the latter mostly emerged from the former);

- `boost::weak_ptr`:
  much like `std::weak_ptr` (the latter mostly emerged from the former);

- `boost::scoped_ptr` and `boost::scoped_array`:
  close to `std::unique_ptr` but in two variants to provide different destructors to do a plain `delete` or a `delete[]` on the pointee.

- `boost::intrusive_ptr`:
  similar in purpose to `boost::shared_ptr` or `std::shared_ptr` but storing the rerefence count inside the pointee (which therefore must be accessible).

Boost: Scoped Pointer

A Scoped Pointer considers itself to be the sole owner of a

- **single object allocated on the heap**

and will finally destroy its pointee (if any) when going out of scope or a new pointee is assigned.

- There is no copy-constructor and -assignment, the only way to re-assign a `boost::scoped_ptr` is via `swap` (available globally and as member function);

- Final destruction will use `delete`, therefore expecting the pointee is a single object.

> **[!]** If a `boost::scoped_ptr` is initialized with an address not returned from `new` or is pointing to an array of objects returned from array heap allocation, undefined behaviour will result at deletion time.

Boost: Scoped Array

A Scoped Array considers itself to be the sole owner of

- **an array of objects allocated on the heap**

and will finally destroy its pointee (if any) when it goes out of scope or a new pointee is assigned.

- There is no copy-constructor and -assignment, the only way to re-assign a `boost::scoped_array` is via `swap` (available globally and as member function);
- Final destruction will use `delete[]`, therefore expecting the pointee is an array of objects.

> **[!]** If a `boost::scoped_array` is initialized with an address not returned from `new` or pointing to a single object returned from plain heap allocation, undefined behaviour will result at deletion time.

Boost: Intrusive Pointer

An Intrusive Pointer is much like a reference counted `boost::shared_ptr` or `std::shared_ptr`.

- Instead of allocating reference counts separately it expects two global functions overladed for pointers to the pointee's type:
    - `intrusive_ptr_add_ref` – called when a new referrer for the pointee is added.
    - `intrusive_ptr_release` – called when an existing referrer of the pointee gets re-assigned or goes out of scope.
- Furthermore there is a class `boost::intrusive_ref_counter` from which the pointee's class may be derived.*

> Boost recommends in case of doubt to prefer ordinary shared pointers and to avoid using intrusive pointers without good reason.

---

*: Of course given its source is written from scratch or at least available and can be modified.

Intrusive Pointer Example

To make `MyClass` usable with intrusive pointers it can be written as:*

```
class MyClass :
    public boost::intrusive_ref_counter<
                     MyClass,
                     boost::thread_unsafe_counter
           > { … };
```

Or if it should be usable in a multi-threaded environment:

```
class MyClass :
    public boost::intrusive_ref_counter<
                     MyClass,
                     boost::thread_safe_counter
           > { … };
```

Then there can be intrusive pointers of `MyClass`:

```
boost::intrusive_ptr<MyClass> p{new MyClass(…)};
```

*: The second template argument may be omitted as it defaults to `boost::thread_unsafe_counter`.

Boost: Pointer Container

A number of Pointer Containers has been made available by boost, paralleling the STL containers with a pointer version that

- omits the pointer syntax at instantiation,
- adds one level of dereferencing to each member access, and
- considers its elements as pointers owning the memory pointed to.

While the former two are more a matter of convenience (see example on next slide), the last one has a severe semantical implication:

- If the container goes out of scope it deletes all the pointees of its (still) contained elements.*

> **[!]** Storing non-owning pointers or pointers that do not even point to heap-allocated memory in a pointer container will cause undefined behaviour when the container goes out of scope.

---

*: This effect can alternatively be achieved by storing `std::unique_ptr`-s in an ordinary container.

Pointer Container Example

Storing and later on processing a `boost::ptr_vector`:*

```
boost::ptr_vector<MyClass> v;
…
// fill in some content (probably in a loop):
… v.push_back(new MyClass(…));

// process later (or maybe in a different thread):
while (!v.empty()) {
   … v.back() …; // access MyClass as a whole
   … v.back().m …; // access MyClass data member
   … v.back().f(); // call MyClass member function
   v.pop_back();
}
```

In case the processing loop is not reached or left before the content is fully processed, **the pointer container destructor** will call `delete` for the pointers still contained, avoiding a memory leak.

*: If a pointer container actually gets filled and processed concurrently as suggested by the comment in the example, mutexes or other synchronization techniques must be added as modifying operations are not thread-safe by themselves.

Pointer Container Substitute

Storing and later on processing a container of custodial pointers:*

```
std::vector<std::unique_ptr<MyClass>> v;
…
// fill in some content (probably in a loop):
… v.emplace_back(new MyClass(…));

// process later (or maybe in a different thread):
while (!v.empty()) {
    … *(v.back()) … // access MyClass as a whole
    … v.back()->m … // access MyClass data member
    … v.back()->f() … // call MyClass member function
    v.pop_back();
}
```

In case the processing loop is not reached or left before the content is fully processed, **the `std::unique_ptr` destructors** will call `delete` for their pointees, avoiding a memory leak.

*: If an STL container actually gets filled and processed concurrently as suggested by the comment in the example, mutexes or other synchronization techniques must be added as modifying operations are not thread-safe by themselves.

Garbage Collection

There is no garbage collection in C++ because of a specific difficulty:

- **An address once obtained from `new` may not be visible in any memory location capable of holding a heap address**, instead it

    1. may have been modified by address arithmetic … which will of course be reverted before the `delete` takes place;

    2. may be temporarily stored in an integral type* … and will of course be restored to a pointer of appropriate type before the `delete` takes place.

Both are not a sign of bad programming style but have some valid uses in the C and C++ code base written in the last 35 years, so they cannot be easily ruled out by a new language standard.

*: C/C++ even guarantees that when an integral type of sufficient size is used as temporary to store a pointer, after assigning the content back to the original pointer type the memory location pointed to will not have changed … which by no means says that the bit patterns stays the same all the time!

C++11: Garbage Collection API

C++11 has defined an Garbage Collection ABI to enable *Interested Third Parties* to supply garbage collection as add-on library.*

Mainly the ABI allows to say (put colloquially):

- **The object at this address I name to you may appear not to be any longer in use.**
    - You may not find it in any memory location capable of holding a heap pointer. Nevertheless be assured: it *is* still in use, so *do not garbage collect it*, I'll take up responsibility and return the reserved space in due course when its *really* not in use any more.
- **In this memory area I name to you, you may find storage cells looking like pointers to heap memory, but they aren't.**
    - So, in case there is any memory pending to be freed and *its only use appears to be from inside this area*, feel free to go ahead and garbage collect that stuff.

---

*: It will surely be interesting to watch such efforts and whether any third-party garbage collector for C++ gets into wide-spread use. If so, then probably rather for new software, not for large amounts of legacy code (including libraries), and maybe only with additional support by compiler warnings.