C++ London University Session 10

Tristan Brindle

Feedback and Communication

- Your feedback is vital
- Otherwise, we don't know what you don't know!
- Please join the #cpplondonuni channel on the cpplang Slack — Go to https://cpplang.now.sh/ for an "invitation"

Today's Lesson Plan

- Object lifetime in C++
- The stack and the heap
- Smart pointers and when to use them

Object Lifetime in C++

- Object lifetime refers to the time between the creation of an object and its destruction

- Unlike many other languages, C++ does not have *garbage collection* — managing the lifetime of objects is the responsibility of the programmer

- There are four kinds of object lifetime (AKA “storage duration”) in C++: *automatic*, *static*, *thread local* and *dynamic*

Automatic Lifetime

- So far we have only dealt with objects with *automatic* storage duration
- These are “normal” local variables used in functions
- Their lifetime extends from the point of declaration until the enclosing scope is exited

Static and Thread Lifetime

- Objects can also have *static* or *thread* storage duration

- The lifetime of these objects runs from the point of definition until the termination of the program (*static* lifetime) or the termination of the current thread (*thread* lifetime)

- Local variables (inside functions) with *static* lifetime are declared with the `static` keyword. Objects with *thread* lifetime are declared with the `thread_local` keyword.

- Beware the different meanings of the `static` keyword!

Dynamic Lifetime

- Finally, objects can have *dynamic* lifetime
- Objects with dynamic lifetime are created with the `new` keyword, and are destroyed with `delete`
- Arrays with dynamic lifetime are created with `new[]`, and destroyed with `delete[]`
- Golden, inviolable rule: every `new` must be paired with a `delete`, and every `new[]` must be paired with a `delete[]`

Stack and Heap

- Local variables with automatic storage duration are allocated in a memory area called the *stack*
- The stack is relatively small (8MB on Linux), and exhausting it will crash your program!
- Every thread of execution has its own stack
- Stack allocation is very fast, but its limited size means we can’t use it to store large objects

Stack and Heap

- Unlike automatic objects, dynamic objects are allocated on the *heap*, otherwise known as the free store

- The size of heap-allocated objects is (in principle) limited only by the amount of RAM on your system

- Heap allocation is much slower than stack allocation, as we need to call out to the operating system to request more memory

Pros and Cons

- Automatic/stack variables:
    - Pros: Very fast to create, compiler manages lifetime
    - Cons: Limited size, exhaustion crashes program
- Dynamic/heap variables:
    - Pros: “unlimited” memory available for our program, necessary for runtime polymorphism
    - Cons: Slower to allocate, need to manage memory manually using new and `delete`

Perilous Programming Pitfalls

- Failing to delete an object created with new or new[] is a memory leak (bad)
- Deleting an object twice, or mixing delete and delete[], can cause your program to crash (very bad)
- Ensuring that your program calls the right form of delete, exactly once, exactly in the right place, is a very very difficult task — particularly when exceptions are involved.
- But automatic variables are wonderful: the compiler will ensure they are destroyed, no matter how we exit the scope
- Wouldn’t it be great if we could use an automatic object to “manage” the lifetime of a dynamic object…?

Managing dynamic lifetime

- Let’s say we have a class Student, and we want a class Form which holds an array of N Students
- We could do this by dynamically allocating an array using operator new[] in the Form constructor, and calling delete[] in the Form destructor…
- …and handle copy construction and copy assignment…
- …and move construction and move assignment in C++11…
- …and worry about getting everything right in the presence of exceptions
- Better: use std::vector!

Managing dynamic lifetime

- Behind the scenes, std::vector deals with allocating, copying, resizing, deleting etc a dynamic array
- std::vector is a great example of using an automatic object to manage the lifetime of a dynamic object
- This is the principle behind smart pointers!

Smart Pointers

- A smart pointer is an automatic object that behaves like a raw pointer, but manages the lifetime of a dynamic object for us

- In C++11, the standard library has two smart pointers we can use: unique_ptr and shared_ptr

- (C++98 had an earlier attempt at a smart pointer, called auto_ptr: do not use it, ever!).

Exercise

- Let’s write a simple smart pointer!
- https://github.com/CPPLondonUni/simple_smart_pointer

Ownership

- Conceptually, the *owner* of an object is whichever part of our program is responsible for deleting it

- Operator new returns a raw pointer to the dynamic object. When we pass raw pointers around, it can become very difficult to keep track of who owns the object

- Smart pointers are a compiler-assisted method of managing object ownership

std::unique_ptr

- A unique_ptr<T> is a wrapper around a raw pointer, which calls delete in its destructor
- It supports all the usual operations on a pointer: dereference (operator*), member access (operator->), etc
- unique_ptrs cannot be copied: they can only be moved
- In C++14, we can use std::make_unique() to create a unique_ptr

The one-slide introduction to move semantics

- C++11 introduced move semantics
- Move is an optimisation of copy, and leaves the source object in a moved-from state
- An object in a moved-from state generally cannot be used, only assigned to or destructed
- You can move an object by calling std::move(x), for example

```
std::vector<int> v1{1, 2, 3};
std::vector<int> v2 = std::move(v1);
// We have move-constructed v1 from v2
// v2 is now in a moved-from state
```

unique_ptr and polymorphism

- With raw pointers, a base class pointer can point to an instance of a derived class
- When we call delete on the base class pointer, it will call the derived class destructor as long as it is `virtual`
- For example:

```
class Base {
    virtual ~Base() = default;
};

class Derived : public Base {};

Base* ptr = new Derived{};
delete ptr; // Calls ~Derived()
```

unique_ptr and polymorphism

- The same applies to unique_ptr: a unique_ptr<Base> can hold a pointer to a derived class
- But note that we still need a virtual destructor!
- As a side note, this is why unique_ptr cannot be copied: C++ does not have a notion of a “virtual copy constructor”
- Trying to copy a unique_ptr<Base> would not know that it needs to call the Derived copy constructor
- You can get around this by having a virtual clone() member function

std::shared_ptr

- unique_ptr models unique ownership — the clue is in the name!
- By contrast, shared_ptr models shared ownership, via reference counting
- Use std::make_shared() to create a shared_ptr

Reference Counting

- When we create a shared_ptr, its initial reference count is 1
- When we copy a shared_ptr, its reference count is incremented
- When we destroy a shared_ptr, its reference count is decremented
- When the reference count is zero, it means that the last object has been destroyed and the object can be freed
- For bonus points: what happens to the reference count when we *move* a shared_ptr?

Beware of cycles!

- Reference counting can be thought of as a simple form of garbage collection
- But it’s easy to get into a situation where two objects hold references to each other: this is called a reference cycle
- This prevents the objects from being destroyed — we have a memory leak!

Beware of cycles! (2)

- We can use a std::weak_ptr to break reference cycles
- A weak_ptr is created from a shared_ptr, but does not increase the reference count
- To use the contained value, we must first convert the weak_ptr to a shared_ptr by using the lock() member function
- If the shared_ptr has been destroyed (the last strong reference has been released), then lock() will return an empty object

Smart pointer guidelines

- In C++14, NEVER use “naked” new or delete. ALWAYS use smart pointers or containers!
- unique_ptr should be your default, go-to smart pointer class
- Use shared_ptr rarely, and only when shared ownership is genuinely required
- Use raw pointers (only) to represent unowned dynamic objects: a raw pointer means “I don’t have to delete this”.

Online Resources

- https://isocpp.org/get-started
- cppreference.com — The bible, but aimed at experts
- cplusplus.com — Another reference site, also has a tutorial section
- learncpp.com — Free online tutorial, very up-to-date
- https://www.pluralsight.com/authors/kate-gregory - Comprehensive set of courses from an experienced C++ trainer (free trial)
- reddit.com/r/cpp_questions
- Cpplang Slack channel — https://cpplang.now.sh/ for an “invite”
- StackOverflow (but…)

Thanks for coming!

C++ London University:

- Website: cpplondonuni.com
- Twitter: @cpplondonuni
- Github: github.com/CPPLondonUni

Where to find Tom Breza:

- On Slack: cpplang.slack.com #learn #cpplondon
- E-mail: tom@PCServiceGroup.co.uk
- Mobile: 07947451167

My stuff:

- Website: tristanbrindle.com
- Twitter: @tristanbrindle
- Github: github.com/tcbrindle

See you next time! 🙂