Introduction to Object Orientated Programming in C++ — Session 4

Tristan Brindle

Register now for C++ on Sea!

https://cpponsea.uk/

Feedback

- We love to hear from you!

- The easiest way is via the *cpplang* group on Slack — we have our own channel, *#cpplondonuni*

- Go to https://cpplang.now.sh/ for an “invitation”

This session

- Dynamic memory management
- Smart pointers
- Virtual destructors

Last week's exercise

- https://github.com/CPPLondonUni/oop_logging_exercise

Interface vs implementation

- A major goal of OOP is to separate *interface* from *implementation*

- By using “interface classes”, we can write code which relies only an interface, without knowing any details of how the implementation works

- This allows us the freedom to change the implementation without having to change (or recompile) any code which only uses the interface

Houston, we have a problem

- Problem: which implementation we want to use may depend on *run-time conditions*

- For example, we might want to use either a `Dog` or `Cat` implementation of `Animal`, depending on the user's preference

```
void do_something(Animal& a);

Animal get_preferred_animal()
{
    auto opts = ask_user();

    if (opts.prefers_cats()) {
        return Cat{};
    } else {
        return Dog{};
    }
}

int main()
{
    Animal a = get_preferred_animal();

    do_something(a);
}
```

Houston, we have a problem

- Solution?: Use references?

```
void do_something(Animal& a);

Animal& get_preferred_animal()
{
    auto opts = ask_user();

    if (opts.prefers_cats()) {
        Cat c{};
        return c;
   } else {
        Dog d{};
        return d;
    }
}

int main()
{
    Animal& a = get_preferred_animal();

    do_something(a);
}
```

(Slightly) better solution

```
void do_something(Animal& a);

Animal* get_preferred_animal()
{
    auto opts = ask_user();

    if (opts.prefers_cats()) {
        return new Cat{};
    } else {
       return new Dog{};
    }
}

int main()
{
    Animal* a = get_preferred_animal();
    do_something(*a);
    delete a;
}
```

Operator new

- The `new` keyword in C++ creates an instance of a type on the *heap* or *free store*

```
int* ptr = new int{3};
```

- `new` returns a *pointer* to the object that was just created

- The `delete` keyword is used to destroy an object created with `new`

```
delete ptr;
```

- Every use of new **must** be paired with a `delete`

New and delete

- Whenever we create an object with `new`, we **must** have a corresponding `delete` called *exactly once*

- If we **do not** call `delete`, we have a *memory leak* (bad)

- If we call `delete` **more than once** with the same pointer, our program will most likely crash (very bad)

- Ensuring that we call `delete` exactly once, exactly when we have finished using the object, is a very hard problem!

Using destructors

- Problem: how do we ensure that every new is paired with a delete, called exactly once?
- Solution (most other languages): Use a garbage collector!
- Better solution (C++): Use destructors!
- The C++ language guarantees that the destructor for an object is called when we leave the scope it is declared in
- By calling `delete` in a destructor, we ensure that it always gets called, no matter what
- Classes which manage dynamic allocations in this way are often called *smart pointers*

Example

```
class BadAnimalPtr {
    Animal* ptr = nullptr;

public:
    BadAnimalPtr() = default;

    BadAnimalPtr(Animal* a)
        : ptr(a)
    {}

    ~BadAnimalPtr() { delete ptr; }

    Animal& operator*() const { return *ptr; }

    // other operators...
};
```

Example (2)

```
void do_something(Animal& a);

BadAnimalPtr get_preferred_animal()
{
    auto opts = ask_user();

    if (opts.prefers_cats()) {
        return BadAnimalPtr{new Cat{}};
    } else {
       return BadAnimalPtr{new Dog{}};
    }
}

int main()
{
    BadAnimalPtr a = get_preferred_animal();
    do_something(*a);
    // delete a;
}
```

- Yay?

Smart pointers

- Problem: what happens when we **copy** a `BadAnimalPtr`?

```
int main()
{
    BadAnimalPtr a = get_preferred_animal();
    BadAnimalPtr b = a;
}
```

- The (default) copy constructor for `BadAnimalPtr` will simply copy the pointer value

- Both variables `a` and `b` will call `delete` in their destructor, with the same pointer value

- When the second destructor runs, the program will crash!

Move semantics

- C++11 added a new fundamental operation for objects, the notion of *moving*

- When we *copy construct* an object, we create a new object *leaving the source unchanged*

- When we *move construct* an object, we create a new object by (potentially) *modifying the source object*

- Similarly, *move assignment* performs an assignment by (potentially) modifying the source object

Move semantics

- You can think of moving as in some sense “stealing” the contents of the source object

- We can mark an object as available for moving by using the `std::move()` function:

```
void func(Moveable m);

int main()
{
    Moveable m;
    func(std::move(m)); // m is *moved* into func
}
```

- The source object is left in a *moved-from state*: generally the only thing we can do with it is to let it be destroyed

std::unique_ptr

- Some classes cannot be copied, only moved — we call these *move-only*
- If we were to make our `AnimalPtr` class move-only, we could avoid the problem with double-deleting!
- Fortunately, the standard library already provides a solution in the form of `std::unique_ptr`

Example

```
void do_something(Animal& a);

std::unique_ptr<Animal> get_preferred_animal()
{
    auto opts = ask_user();

    if (opts.prefers_cats()) {
        return std::unique_ptr<Animal>{new Cat{}};
    } else {
        return std::make_unique<Dog>();
    }
}

int main()
{
    std::unique_ptr<Animal> a = get_preferred_animal();
    do_something(*a);
}
```

Smart pointer guidelines

- Smart pointers should always be used in preference to manual memory management
- **Never*** use raw `new` and `delete` in new code!
- The C++ standard library provides two smart pointers: `unique_ptr` and `shared_ptr`
- Prefer `unique_ptr`; use `shared_ptr` only when necessary
- Raw pointers (`T*`) should never be owning

Virtual destructors

- Problem: a `unique_ptr<Base>` may be created with an instance of a derived class
- When the `unique_ptr`'s destructor calls `delete`, it only passes the base class pointer
- How do we ensure that the derived class's destructor gets called?
- Solution: virtual destructors

Virtual destructors

- Memory management can be tricky to get right when working in OO-style C++
- Since C++11, smart pointers make this very much easier
- For this to work correctly, we need to declare a *virtual destructor* in our base classes:

```
struct Animal {
    virtual void speak() const = 0;

    virtual ~Animal() = default;
};
```

Virtual destructors

- Guideline: if your class has *any other* virtual functions, you should provide a virtual destructor
- This includes pure interface classes!
- A virtual destructor may be defined with `= default` if it does not need to perform any special actions
- Do not declare a destructor as *pure virtual* unless you know what you're doing

Eeek!

- Module “questionnaire”
- https://bit.ly/2QmwlAk

Next time

- Initial C++!

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…)