Computing at Scale

Lecture 4: Classes and Move Semantics

Jacob Merson

January 16, 2024

Logistics

Office Hours

Based on the survey:

- Friday 3-5:30PM (is there anyone who is unable to make this time?)
- By appointment (mersoj2@rpi.edu)

Other Logistics

- Homework 0 Due Today
- Code Review 0 Due 1/23
- 1 Page Project Proposal Due 1/27

CR0 Groups

- **Group 1:** Scott Blender, Jay Gaiardelli, Abhiyan Paudel
- **Group 2:** Fuad Hasan, Kairvi Lodhiya, Mikiel Gica,
- **Group 3:** Bibek Shrestha, Ickbum Kim, Zach Knowlan

Today's Agenda

- Finish CMake
- Classes
- Move Semantics

CMake (finish slides from lecture 2)

Classes

Basics

- Built-in types: int, double, char, etc.
- What about something more complex? e.g., a point in 3D
- Concrete classes can be used to define new user defined types and their behavior
- Abstract classes define interfaces

Why Classes?

Encapsulation

- hide implementation details behind an interface
- maintain class invariants
- abstract classes will present a clear way to separate interface from implementation, more soon
- someone can write code without any understanding of how the implementation works

Terminology

- **object:** instance of a class
- **member function:** function in a class
- **member data:** data in a class
- **stack memory:** memory allocated for local variables at compile time
- **heap memory:** dynamically allocated memory at runtime

Concrete Class

- behave like built in types
- group data and functions that operate on that data
- examples: string, vector, complex number, etc.
- can be constructed in stack memory
- representation is part of definition
- can define operators, note: only do this when semantics are obvious
- list of overloadable operators:
  `https://en.cppreference.com/w/cpp/language/operators`

Concrete Class Example

```
class Point {
public:
  Point(double x, double y, double z);
  // can define operator overloads
  friend bool operator==(const Point& lhs, const Point& rhs);
  // if we have < we can define sort order
  friend bool operator<(const Point& lhs, const Point& rhs);
  // C++20 should use <=>, spaceship operator
private:
  double x_, y_, z_;
};
```

constness

- const member functions cannot modify member data (unless mutable)
- const objects can only call const member functions
- aim to make member functions const whenever possible
- important to check for const correctness in code review

Classes vs Struct

- technically same in C++, struct has public members by default, class has private members by default
- idiomatic C++ uses structs to aggregate data, classes for encapsulation

Object Lifetime

- objects allocated on the stack are destroyed when they go out of scope
- objects allocated on the heap are destroyed when they are deleted

Object Lifetime Example

```
void f() {
  {
    Point p(1, 2, 3);
  // p goes out of scope
  }
  {
    Point* q = new Point(4, 5, 6);
    // q goes out of scope, pointed to memory not deleted
  }
  // error q is out of scope
  delete q;
}
```

Demo

- Write class that prints a message in the constructor and destructor.

RAII

Important C++ idiom: Resource Acquisition Is Initialization

- use constructors to acquire resources, destructors to release them
- ensures resources are released when object goes out of scope
- RAII types include `std::unique_ptr`, `std::shared_ptr`

Rule of 0, Rule of 5

- Rule of 5: if you need a custom destructor, copy constructor, copy assignment operator, move constructor, or move assignment, you need to implement all of them
- Rule of 0: classes that have custom destructors or copy/move constructors should deal solely with ownership

`https://en.cppreference.com/w/cpp/language/rule_of_three`

Copy swap idiom

```
struct A
{
    int n;
    std::string s1;
    A() = default;
    A(A const&) = default;
    // user-defined copy assignment (copy-and-swap idiom)
    // use when need exception gauranty
    A& operator=(A other) // make a copy of A
    {
        std::cout << "copy␣assignment␣of␣A\n";
        std::swap(n, other.n);
        std::swap(s1, other.s1);
        return *this;
    }
};
```

Demo

- Write a class that allocates memory on the heap (basic vector)
- Copy constructor / assignment

Move?!

- Move semantics allow for efficient transfer of resources
- example: large vector, we don’t want to copy it

- **lvalue:** object with a name
- **rvalue:** temporary object
- lvalues can be converted to rvalues with `std::move`

L and R values

- **lvalue:** object with a name
- **rvalue:** temporary object

```
int x = 5; // x is an lvalue
int y = x; // x is an lvalue, y is an lvalue
int z = x + y; // x + y is an rvalue

// unless we have templates
void f(int&& x) // x is an rvalue reference
void g(int& x) // x is an lvalue reference
void h(int x) // x is a lvalue
```

move semantics

- move constructor and assignment operator follow normal function overloading rules
- move constructor and assignment operators take an rvalue reference (&&)
- move operations move resources from the rvalue to lvalue (assuming that's what you teach your type to do)
- default move constructor and assignment operator are generated if you don't define them. These call move on each member
- move operations should be noexcept
- move operations should leave the moved from object in a valid state (class invariants should be maintained)

In class exercise

Using the vector class we wrote earlier, write a move constructor and move assignment operator. You will want to use the reference `https://en.cppreference.com/w/cpp/language/move_constructor`

Question: How do you test if the move or copy constructor is being called?

Abstract Class

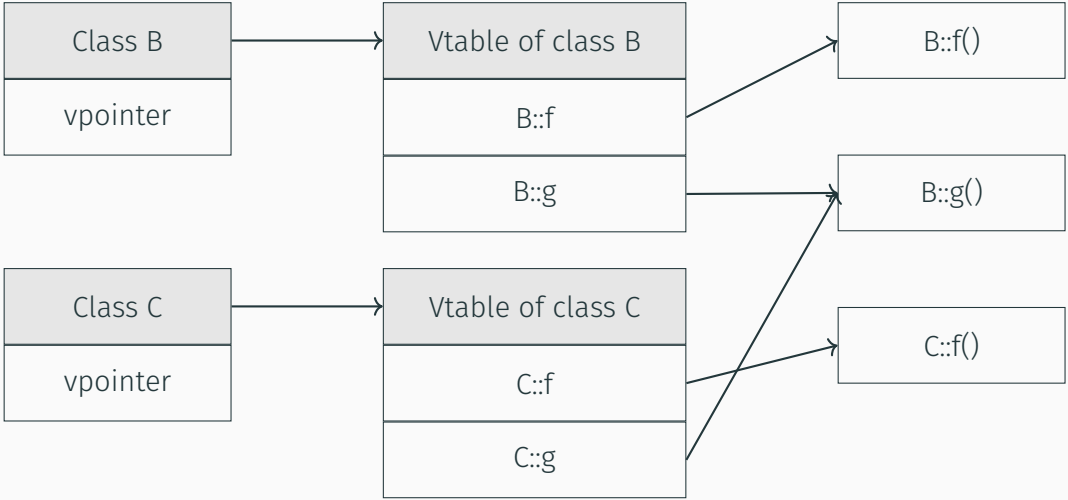
- separates implementation from interface
- implementations must be accessed through pointers or references
- typically allocated on the heap

Derived Class

```
// in header file
class B {
public:
  virtual void f();
  virtual void g();
  // all destructors
      should be noexcept
  virtual constexpr ~B()
      noexcept = default;
};
class C : public B {
  public:
    void f() override;
};
```

```
// in cpp file
// note using "std" prefix
void B::f() { std::cout << "B::f\n"; }
void B::g() { std::cout << "B::g\n"; }
void C::f() { std::cout << "C::f\n"; }
```

vtable


Class B
vpointer
Vtable of class B
B::f
B::g
B::f()
B::g()
Class C
vpointer
Vtable of class C
C::f
C::g
C::f()