VGP336 - OOP Review

Instructor: Peter Chan

Agenda

- Introduction
- Course Syllabus
- OOP Review

Object-Oriented Programming

- C++ introduced the concept of class which gives us the following:
  - Ability to define custom data types
  - Encapsulation
  - Inheritance
  - Polymorphism
- What is a type?
  - All types in C++ have two things
  - How much memory does it need?
  - What operations can you perform?

Object-Oriented Programming

```
class Car
{
public:
     void Move() { pos++; }

private:
     int pos = 0;
     int fuel = 100;
};

Car myCar;
myCar.Move();
```

Encapsulation

- Mechanism which allows hiding information
  - i.e. public, protected, private
- No more “spaghetti” code
  - Global variables are discouraged
  - Well defined code boundary due to class interfaces
- Allows large teams to work together without issues

Inheritance

- Child class will inherit all the member variables and functions from the parent class
- Must use with care; code reuse is not a good reason for inheritance
  - e.g. Cat derving from Car is wrong
- **Inheritance should be used specifically for defining the “is-a” relationship**

Inheritance

When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

Inheritance

```
// Base/Parent class
class Car
{
public:
     void Move() { pos++; }
private:
     int pos = 0;
};

// Derived/Child class
class Tesla : public Car
{
};
```

Inheritance

```
// Tesla is-a Car = good!
class Tesla : public Car
{
};

// Cat is not a Car = bad!! Even though this works!
class Cat : public Car
{
};
```

Inheritance

```
class Car
{
public:
     void Move() { pos++; }
protected:
     int pos = 0;
};
```

```
class Tesla : public Car
{
public:
     // Function overriding - an inheritance feature
     void Move() { pos += turbo ? 10 : 1; }
protected:
     bool turbo = true;
};
```

Inheritance

We have a problem ....

```
Car myCar;          // pos = 0
myCar.Move();       // pos = 1

Tesla myTesla;      // pos = 0
myTesla.Move();     // pos = 10

Car* ptr = &myTesla;
ptr->Move();        // pos = 10 + 1 = 11!!
```

But why? Which Move() to call is decided on the caller's type

Polymorphism

Polymorphism comes to the rescue with the special “magic” via virtual.

```
Car myCar;          // pos = 0
myCar.Move();       // pos = 1

Tesla myTesla;      // pos = 0
myTesla.Move();     // pos = 10

Car* ptr = &myTesla;
ptr->Move();        // pos = 10 + 10 = 20!!
```

But how?

Polymorphism

TODO: Add picture!!!

Operator Overloading

Fancy syntax that allows you to call a function by using an operator symbol

```
class Car
{
public:
     void Move() { pos++; }
     void operator++() { pos++; }
};

Car myCar;
myCar.Move(); // Car::Move(myCar);
myCar++;      // Car::operator++(myCar);
```

Operator Overloading

Why? Sometimes operator overloading gives better syntax

```
class Vector2
{
public:
     void Add(Vector2) { ... }
     void operator+(Vector2) { ... }
};

Vector2 v, u;
v.Add(u);
v = v + u;    // More familiar style in mathematics
```

Initialization vs Assignment

In C++, the distinction between initialization and assignment is particularly important.

```
int n = 42;         // Initialization = 1 step

int n;              // Assignment = 2 steps
n = 42;

// const int i;     // const means assignment is disabled
// i = 7;           // syntax error
```

Initialization vs Assignment

For a class, this means different methods will be called.

```
class Foo
{
public:
     Foo();
     Foo(const Foo&);
     Foo& operator=(const Foo&);
};

Foo a;         // Calls constructor Foo()
Foo b = a;     // Calls copy constructor Foo(const Foo&)
a = b;         // Calls assignment operator=
```

Special Member Functions

In C++, there are six special member functions:

- Constructor
- Destructor
- Copy Constructor
- Assignment Operator
- Move Constructor
- Move Assignment Operator

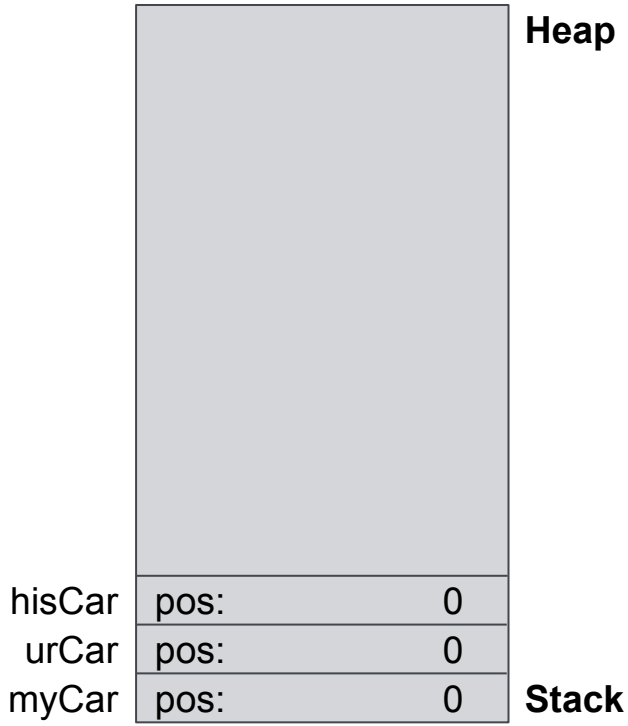
These special member functions may or may not be generated by the compiler for you based on certain rules.

Special Member Functions

```
class Foo
{
public:
    int i = 0;
};

int main()
{
    Foo f;  // Constructor
    std::cout << f.i << "\n";
    f.i = 42;
    std::cout << f.i << "\n";
    Foo g = f;  // Copy constructor
    std::cout << g.i << "\n";
}

// None of the functions are created in Foo, but the code above still works!
```

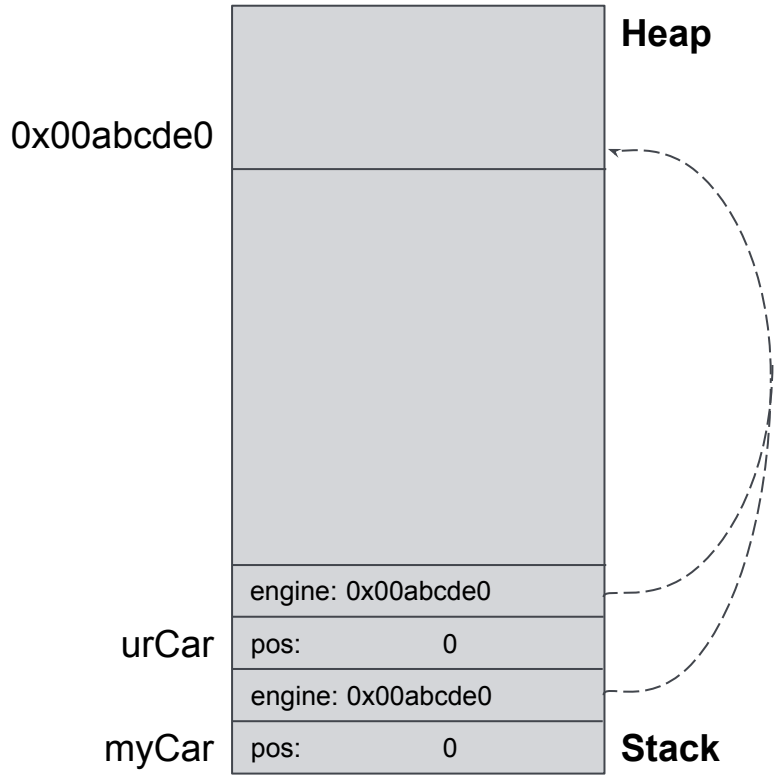
Shallow Copy vs Deep Copy

```
class Car
{
public:
     int pos = 0;
};

int main()
{
     Car myCar;
     Car urCar;
     Car hisCar = myCar;
}
```



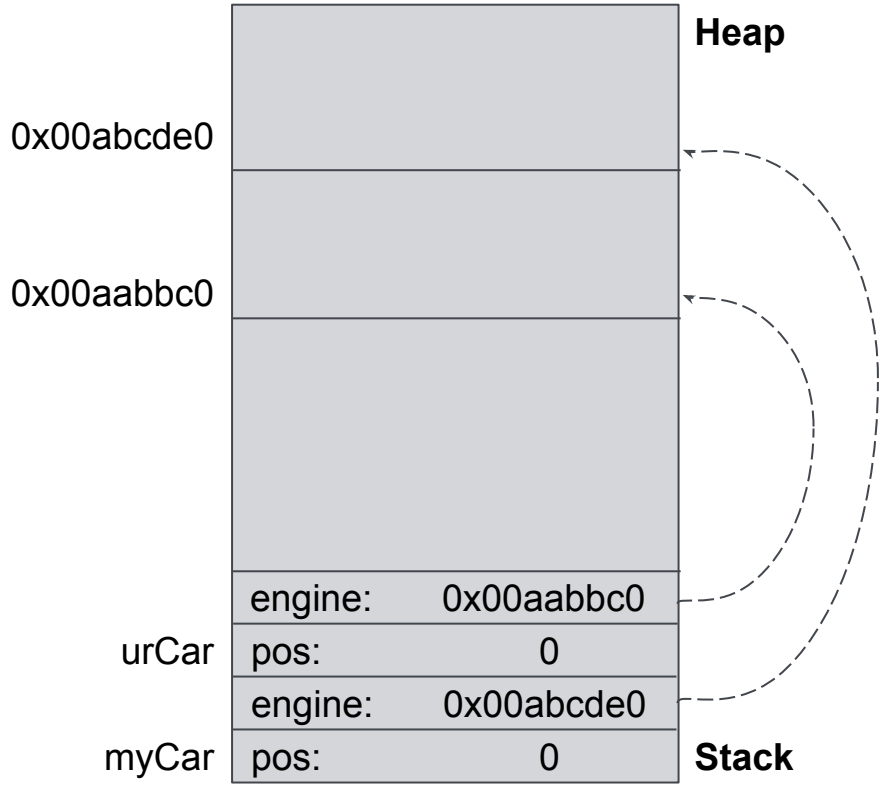
Shallow Copy vs Deep Copy

```
class Car
{
public:
     int pos = 0;
     Engine* engine = nullptr;
};

int main()
{
     Car myCar;
     myCar.engine = new Engine();
     Car urCar = myCar;
}
```



Shallow Copy vs Deep Copy

```
class Car
{
public:
	Car() = default;
	Car(const Car& other)
	{
		pos = other.pos;
		engine = new Engine();
		*engine = *other.engine;
	}
	int pos = 0;
	Engine* engine = nullptr;
};

int main()
{
	Car myCar;
	myCar.engine = new Engine();
	Car urCar = myCar;
}
```



Special Member Functions

Normally, the default created versions will be sufficient already. However, it can be problematic if your class type has pointers or references to memory external to the class.

This is because the generated copy constructor and assignment operator do shallow copy.

To fix it, you will need provide your own versions and perform deep copy operations in them.

Special Member Functions