COMP 3522

Object Oriented Programming in C++

Week 3

Agenda

1. Structs
2. Classes and objects
3. Default constructor
4. Member initialization and default arguments
5. Most vexing parse

COMP 3522

STRUCTS

User-Defined Types: C++ has structs too!

```
struct type_name {
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;

} object_names;
```

Where:

`type_name` is the name for the struct

`object_names` is an optional list of declared objects

User-Defined Types: C++ has structs too!

```
struct product{
     int weight;
     double price;
};

struct product{
     int weight;
     double price;
} apple, banana, melon;
```

CLASSES AND OBJECTS

OOP in C++ (finally!)

- Let's review some fundamental OOP concepts:
  - **Encapsulation**
  - **Abstraction**
  - **Inheritance**
  - **Polymorphism**

Encapsulation

- Process of combining data members & functions into a single unit called class
    - **make data members private**
    - **create public getter/setter functions**

```cpp
class Encapsulation
{
    private:
        int x;

    public:
        void set(int a)
        {
            x = a;
        }

        int get()
        {
            return x;
        }
};
```

```cpp
// main function
int main()
{
    Encapsulation obj;

    obj.set(5);

    cout << obj.get();
    return 0;
}
```

Abstraction

- Only **show relevant details** to user and **hide irrelevant details**
- Abstraction in class using access specifiers
  - **public**, **protected**, **private**
- Abstraction in header files
  - ie: `pow()` function in `math.h`
  - Don't know how `pow` implemented in `math`, we just use it
  - `cout << pow(7,3);  //seven to the power of three = 343`

Abstraction

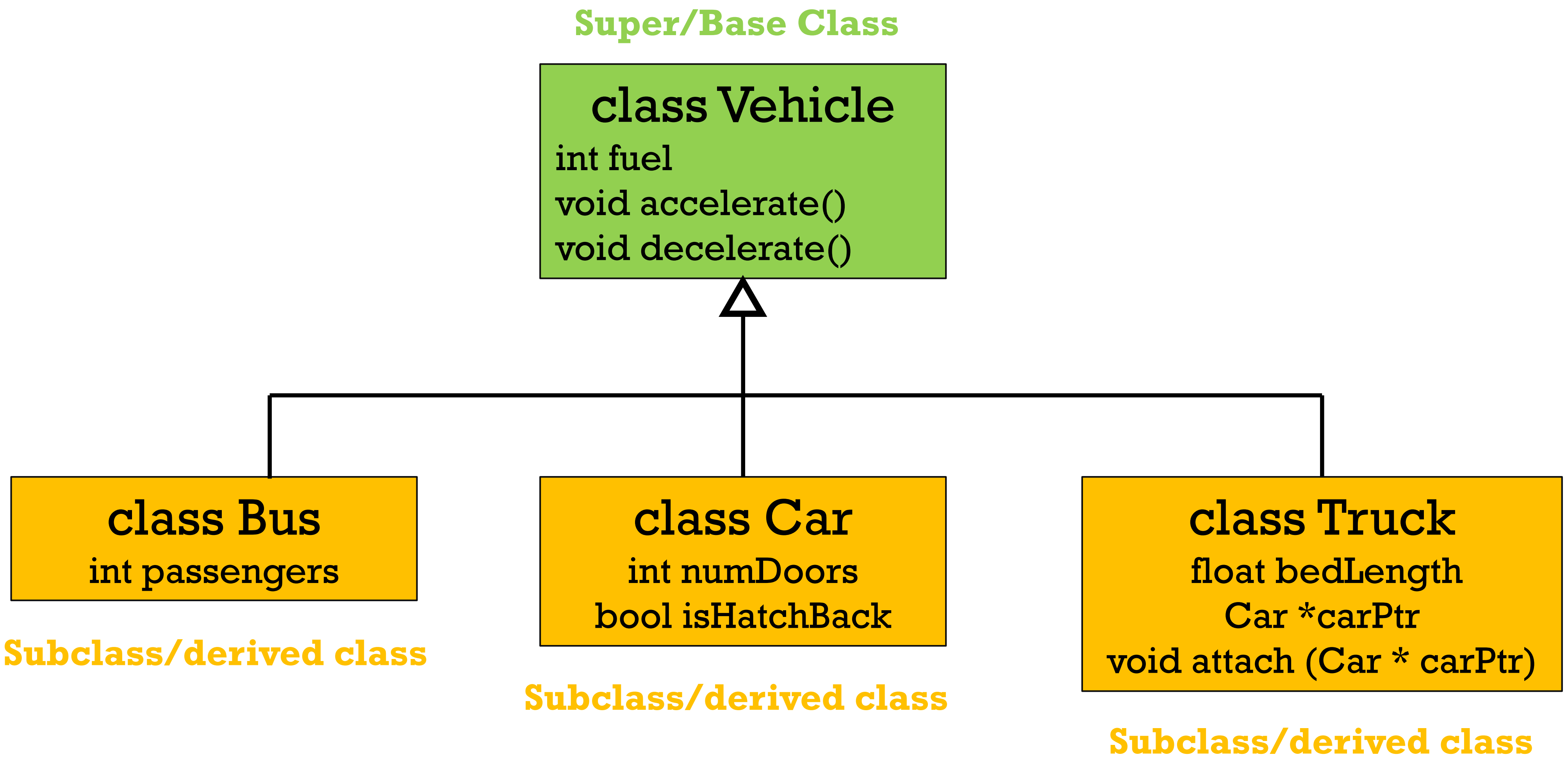
```
class ImplementAbstraction
{
    private:
        int a, b;

    public:
        void set(int x, int y)
        {
            a = x;
            b = y;
        }

        void display()
        {
            cout<<"a = " << a << endl;
            cout<<"b = " << b << endl;
        }
};
```

```
int main()
{
    ImplementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

Inheritance

- Ability of a class to derive properties and characteristics from another class
- **Super/Base Class** – class whose properties inherited by subclass
- **Subclass/derived class** – the class that inherits properties from another class

Inheritance



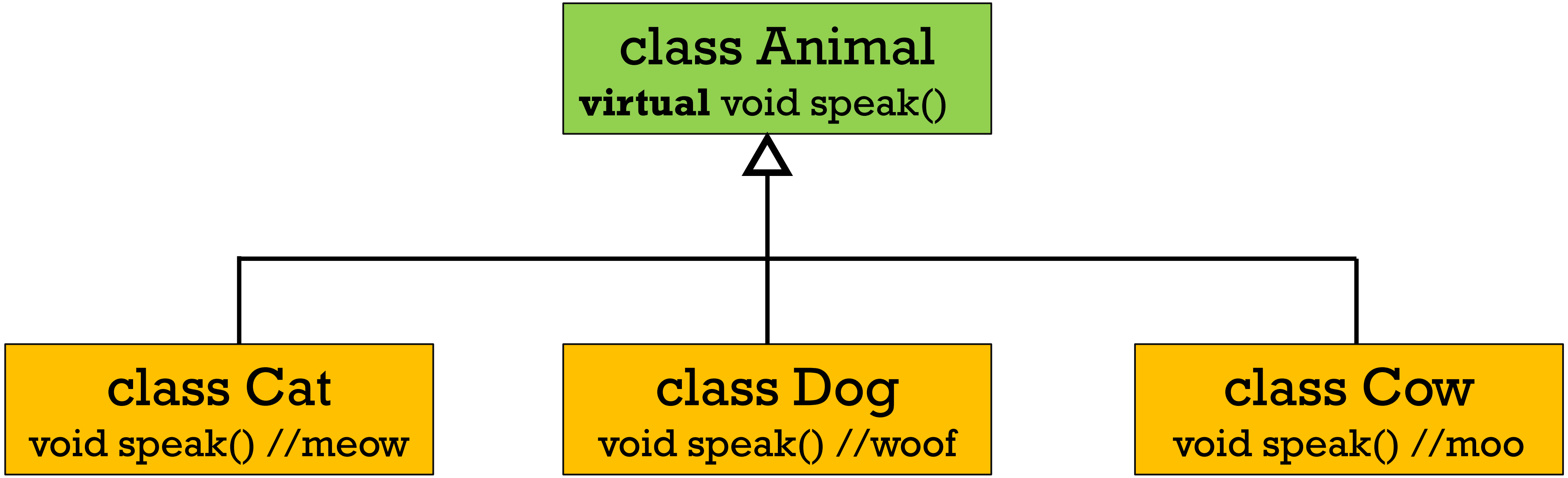
Inheritance

```
//Base class
class Vehicle
{
    public:
        int fuel;
        void accelerate();
        void decelerate();
};

// Sub class inheriting from Base Class(Parent)
class Bus : public Vehicle
{
    public:
        int passengers;
};
```

Polymorphism

- Having many forms
- Call to a member function will cause different functions to be executed depending on the type of object invoked

Polymorphism



The C++ class

```
class Animal {
public:
    virtual void speak() {
        cout << "???" << endl;
    }
};

class Cat : public Animal {
public:
    void speak() {
        cout << "meow" << endl;
    }
};
```

…similar code for Cow and Dog

```
Cat cat;
Dog dog;
Cow cow;
Animal *a;

a = &cat;
(*a).speak(); //meow
a->speak(); //meow
a = &dog;
a->speak(); //woof
a = &cow;
a->speak(); //moo
```

The C++ class

- Defined using keyword **class** or **struct**
- A class defines a new data type that can contain:
  1. **Data** referred to as member variables or data members
  2. **Functions** referred to as member functions or (rarely) methods
  3. **Type definitions**
  4. **Contained classes**

C++ Accessibility

- Class members (data and functions) have visibility (just like Java!)
  - **public** members are accessible anywhere
  - **protected** members are accessible in the class and its subclasses (in C++ we call these derived classes)
  - **private** members are only accessible from within the class
- **By default, all class members have private access**
- Note: a struct and a class are the same thing in C++. Except when we use the keyword "**struct**", **members get public access by default**!

Class example: Circle (part 1). **Circle.hpp**

```
class Circle
{
    private:
        double radius;
    public:
        void set_radius(int);
        double area();
};
```

Class example: Circle (part 2). **Circle.cpp**

```
void Circle::set_radius (int new_radius)
{
    radius = new_radius;
}

double Circle::area()
{
    return 3.14 * radius * radius;
}
```

Circle.hpp

```cpp
class Circle
{
    private:
            double radius;
    public:
            void set_radius(int);
            double area();
};
```

Circle.cpp

```cpp
void Circle::set_radius (int new_radius)
{
      radius = new_radius;
}

double Circle::area()
{
      return 3.14 * radius * radius;
}
```

Circle.hpp

```
class Circle
{
    private:
        double radius;
    public:
        void set_radius(int);
        double area();
};
```

Circle.cpp

```
void Circle::set_radius (int new_radius)
{
    radius = new_radius;
}

double Circle::area()
{
    return 3.14 * radius * radius;
}
```

Class example: Circle (part 3)

```
//main.cpp
Circle my_first_circle;
my_first_circle.set_radius(2);
cout << my_first_circle.area() << endl;
```

We can also do this:

```
//Circle.hpp
class Circle
{
        double radius;
    public:
        void set_radius(int);
        double area()
            {return 3.14 * radius * radius};
};
```

DEFAULT CONSTRUCTOR

Where's the constructor?

- We should probably add a **constructor** to our Circle class:

```
//Circle.hpp
class Circle
{
        double radius;
    public:
        Circle(int); // No return type
        void set_radius(int);
        double area();
};
```

Where's the constructor?

- Don't forget to implement the constructor function

```
//Circle.cpp
Circle::Circle(int r)
{
    radius = r;
}
```

But now that we have a constructor…

We **can’t** do this anymore:

```
Circle constructed_with_default_ctor;
```

The compiler will complain that we don’t have a default constructor

Let's overload our constructor!

```
class Circle
{
        double radius;
    public:
        Circle(); // No return type
        Circle(int); // No return type
        void set_radius(int);
        double area(void);
};
```

And add this…

```
//Circle.cpp
Circle::Circle ()
{
    radius = 10; // Magic numbers are bad
                 // But this is a lecture
}
```

We should use “member initialization”

Constructor without member initialization

```
Circle::Circle(int r)
{
    radius = r;
}
```

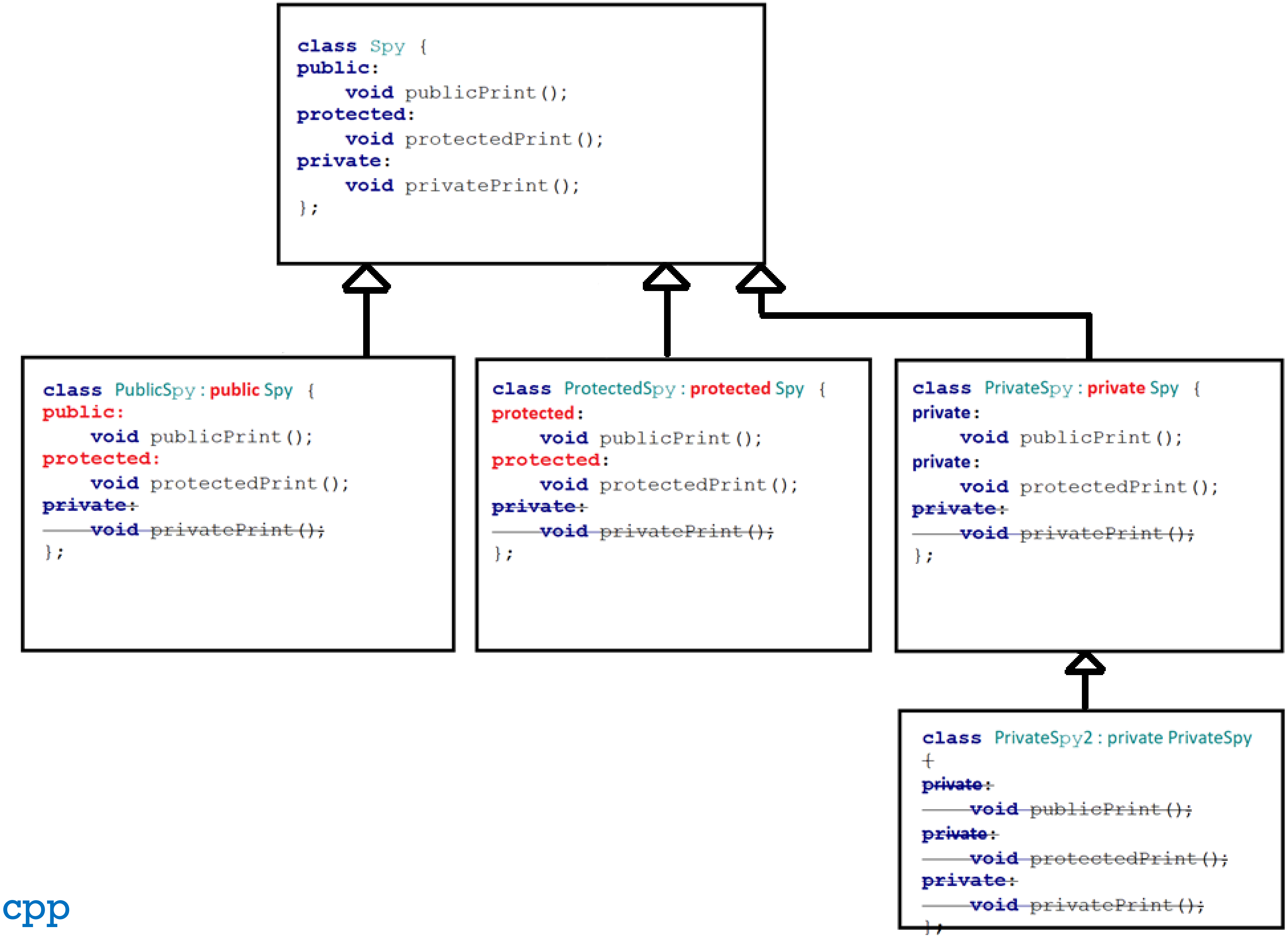
Constructor WITH member initialization

```
Circle::Circle(int r) : radius(r)
{
    // Empty if there’s nothing
    // else to do
}
```

Be careful!

```
Circle my_circle; // Calls the default ctr

Circle my_circle(); // This is a function
                    // prototype. MOST VEXING PARSE

Circle my_circle{}; // Calls default ctr
```

Circle.cpp



Visibility.cpp

MEMBER INITIALIZATION & DEFAULT ARGUMENTS

Did someone say Complex numbers?

- Suppose we have a class representing a Complex number
- A Complex number has two parts:
  1. Real (r)
  2. Imaginary (i)

```
class Complex
{
  private:
    double r, i;
  …
```

The Complex number constructor

Here's a good first pass at the **constructor**:

```
public:
   Complex(double rnew, double inew)
   {
     r = rnew;
     i = inew;
   }
   …
```

There's a problem, though

- The compiler wants to ensure that all member variables are initialized
- It generates a call to the default constructor for the members we don't initialize ourselves:

```
public:
  Complex(double rnew, double inew) : r(), i()
  {
    r = rnew;
    i = inew;
  }
  …
```

This doesn’t always work

- For simple arithmetic types like int and double, it doesn’t really matter if we set their value in an initialization list or in the constructor body
    - Data members of fundamental types that do not appear in the initialization list remain uninitialized
- **<u>There’s a problem with classes though:</u>**
    - A member data item of a class type is implicitly default-constructed if it is not contained in the initialization list
    - In other words, the default constructor is called on class types if they’re not initialized in the initialization list

The Complex number constructor part 1

We should always use the special C++ syntax called the **member initialization list**:

```
class Complex
{
  private:
    double r, i;
  public:
    Complex(double rnew, double inew) : r(rnew), i(inew)
    {
     …
    }
```

The Complex number constructor part 2

In C++, we can use the same identifiers for the constructor parameters and the class members:

- Names in the initialization list outside the parentheses refer to the members
- Inside the parentheses the names follow the scoping rules for a member function (names local to the member function including argument identifiers hide names from the class)

```
  …
private:
  double r, i;
public:
  Complex(double r, double i) : r(r), i(i) { }
  …
```

The Complex number constructor part 3

Let’s create a second constructor where we set the imaginary part of the Complex number to 0

```
public:
   Complex(double r, double i) : r(r), i(i) { }
   Complex(double r) : r(r), i(0) { }
```

The Complex number constructor part 4

We probably want a **default** constructor too:

```
public:
  Complex(double r, double i) : r(r), i(i) { }
  Complex(double r) : r(r), i(0) { }
  Complex() : r(0), i(0) { }
```

Too much! How can we simplify this?

Default arguments!

We can reduce code duplication and complexity by including **default arguments** in the declaration

```
public:
  Complex(double r = 0, double i = 0)
    : r(r), i(i) { }
  …

Complex c;
Complex c1(5);
Complex c2(5,6);
```

Default arguments

- Can be provided **for trailing arguments only**:

```
int f(int, int = 0, char * = nullptr); // OK
int g(int = 0, int = 0, char *);       // ERROR
int h(int = 0, int, char * = nullptr); // ERROR

// Space between * and = is needed!
int creates_error(char *= nullptr);    // ERROR
```

C++ constructor style note

- **Data members MUST be initialized in the order in which they are declared in the class**
- The compiler may emit a warning if we don't respect this recommendation
- In C++ the **order of class/struct member initialization is determined by the order of member declaration** and not by the order of their appearance in member initialization list.
- Avoid generating this warning

https://stackoverflow.com/questions/24285112/why-must-initializer-list-order-match-member-declaration-order

Default constructor (a close analysis)

- A constructor
  - no arguments, or has default values for every argument
- Not mandatory, but we should define one whenever possible
  - It is cumbersome (as we will see) to implement containers (lists, trees, matrices) of types that don't have default constructors
  - Eliminates the possibility of uninitialized variables of a type
  - Variables initialized in an inner scope that exist for algorithmic reasons in an outer scope must already be constructed with a meaningful value

Ask yourself: does this type have a 'special' value or state we can 'naturally' use as a default?

And just to make things more exciting

- We can also assign default values to member variables
- When we do this, we only need to set values in the constructor that are different from the defaults
- The benefit is more pronounced in large classes

```
class Complex
{
  private:
    double r = 0.0, i = 0.0;
    …
```

Member functions can be const

- We can add the **const** specifier to a member function prototype
- Specifies that the member function does not modify the object for which it is called
- Compiler will catch accidental attempts to violate this promise
- We should always use this with getters, for example:

```
double Cat::get_weight_grams() const
{
      return weight_grams;
}
```

"I promise this function's code will NOT change this object's member variables"

Organizing our code

- Each unit of source code is typically split into:
  - Header file with declarations (.h or .hpp)
  - Source file (.cpp)
- The header file contains declarations of functions and **classes**
- Declarations tell the compiler that the code for the functions signatures exists somewhere and that they can be called in the current compilation unit
- The source file contains the definitions (implementations) of the functions and **classes** declared in the header file

Q: Where do default argument values go?

In the function prototype in the header file

```
// Header file
void f(int x = 1, int y = 2);

// Source file
void f(int x, int y) { … }
```

Agenda

1. Copy constructor
2. Destructor
3. Forward Declaration
4. Inheritance
5. Polymorphism and virtual functions

COMP 3522

COPY CONSTRUCTOR

Speaking of constructors… Copy constructor!

- New concept (not in Java or C)
- There is a shortcut in C for copying objects
- We can define a **copy constructor**

```
class Complex
{
  public:
    Complex(const Complex& c) : r(c.r), i(c.i) {}
    …
```

Copy constructor 4c

Stick with const and & for copy constructor

```
Complex(const Complex& c) : r(c.r), i(c.i) {}
```

const - allows copying mutable AND immutable objects
& - prevents infinite internal copy loops

Speaking of constructors… Copy constructor!

```
//assuming complex class exists


//main.cpp
Complex c;
Complex copyC(c); //COPY c to copyC
Complex anotherCopyC = c; //COPY c to anotherCopyC
anotherCopyC = c //NO COPY CONSTRUCTOR CALL! Calls
assignment operator
```

Copy constructor 2

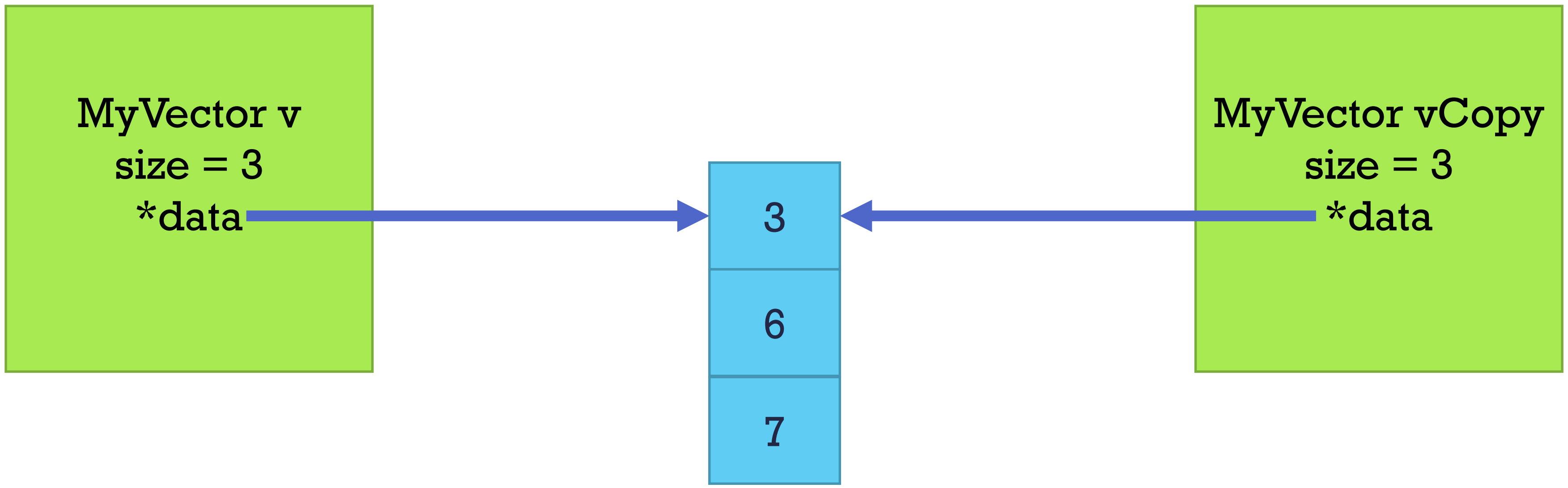
- Compiler will generate one in a standard way that calls the copy constructors of all members in the order of definition
- Use the default if we are just copying all the members:
  - Less verbose
  - Less error-prone
  - Other developers know what our copy constructor does without reading our code
  - Compilers might find optimizations
- **PROBLEM – shallow copy**

Copy constructor example

```
class MyVector
{
  private:
    unsigned size;
    double *data;
  public:
     //didn't specify copy constructor so using default copy constructor - SHALLOW COPY
…
```

Copy constructor example – default shallow

```
//main.cpp
 double vArray[] = {3,6,7};
 MyVector v;
 v.vector_size = 3;
 v.data = vArray;
 vector vCopy = v; //copy v to vCopy
```

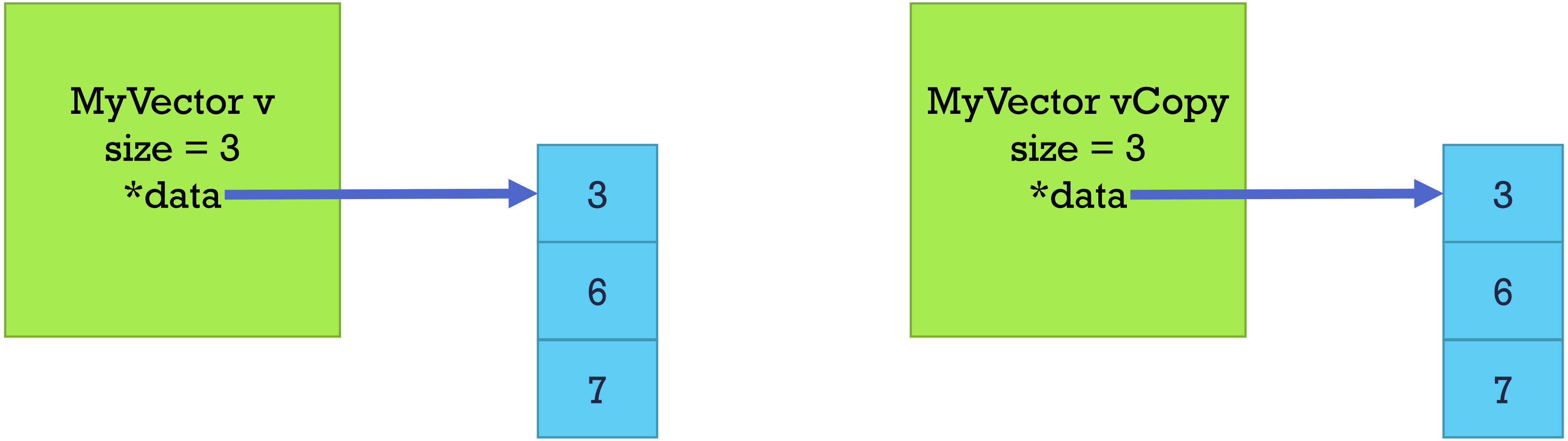


Copy constructor example

```
class MyVector
{
  private:
    unsigned size;
    double *data;
  public:
    MyVector(const MyVector& v) : size(v.size), data(new double[size])
    {
      for (unsigned i = 0; i < size; ++i)
        data[i] = v.data[i];
    }
  …
```

Copy constructor example – not shallow

```
//main.cpp
 double vArray[] = {3,6,7};
 MyVector v;
 v.vector_size = 3;
 v.data = vArray;
 MyVector vCopy = v; //copy v to vCopy
```



myVec.cpp

DESTRUCTOR

Standard C++ class member functions

So far:

1. Default constructor
2. Copy constructor

Next:

The **destructor**.

Destructor

- Member function (of a class)
- **Purpose: to free resources the object acquired during its lifetime**
- Invoked when the lifetime of an object ends
  - Program termination (for statics)
  - End of scope
  - Explicitly call delete, delete[]

Destructor

- The destructor is the complementary operation of the default constructor
- It uses the notation for the complement: **~**

```
class Complex
{
      public:
      ~Complex() { cout << "Destroyed! << endl; }
      …
}
```

Destructor implementation rules

We will return to these in the next few weeks (remind me to tell you why!):

1. **Never throw exceptions from a destructor** (we will learn about C++ exceptions soon!)
2. **If a class contains a virtual function, the destructor should be virtual too** (we will talk about inheritance soon!)

Destructor example

```
class MyVector
{
    public:
        ~MyVector() { delete[] data; }
    …
    private:
        unsigned vector_size;
        double * data;
};
```

FORWARD DECLARATION

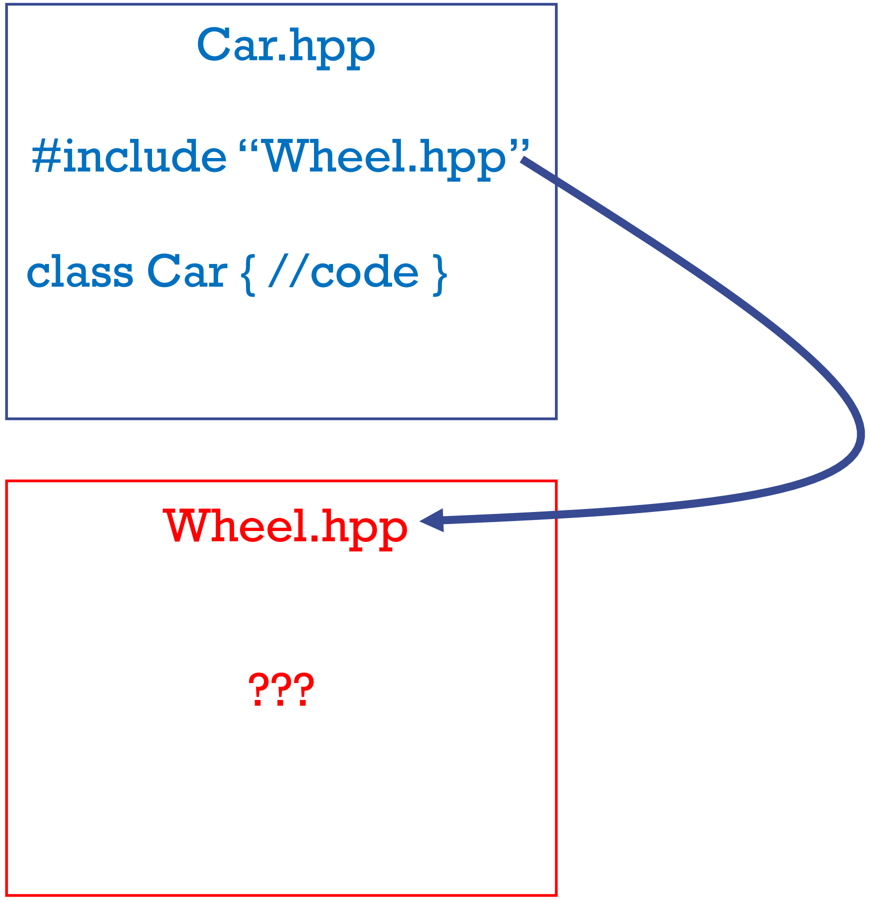
Forward declaration (motivation) (1 of 3)

- Our first C++ OOP conundrum

- Suppose we have a Car class and a Wheel class
  1. A Car has Wheels
  2. A Wheel has a pointer to the Car that possesses it

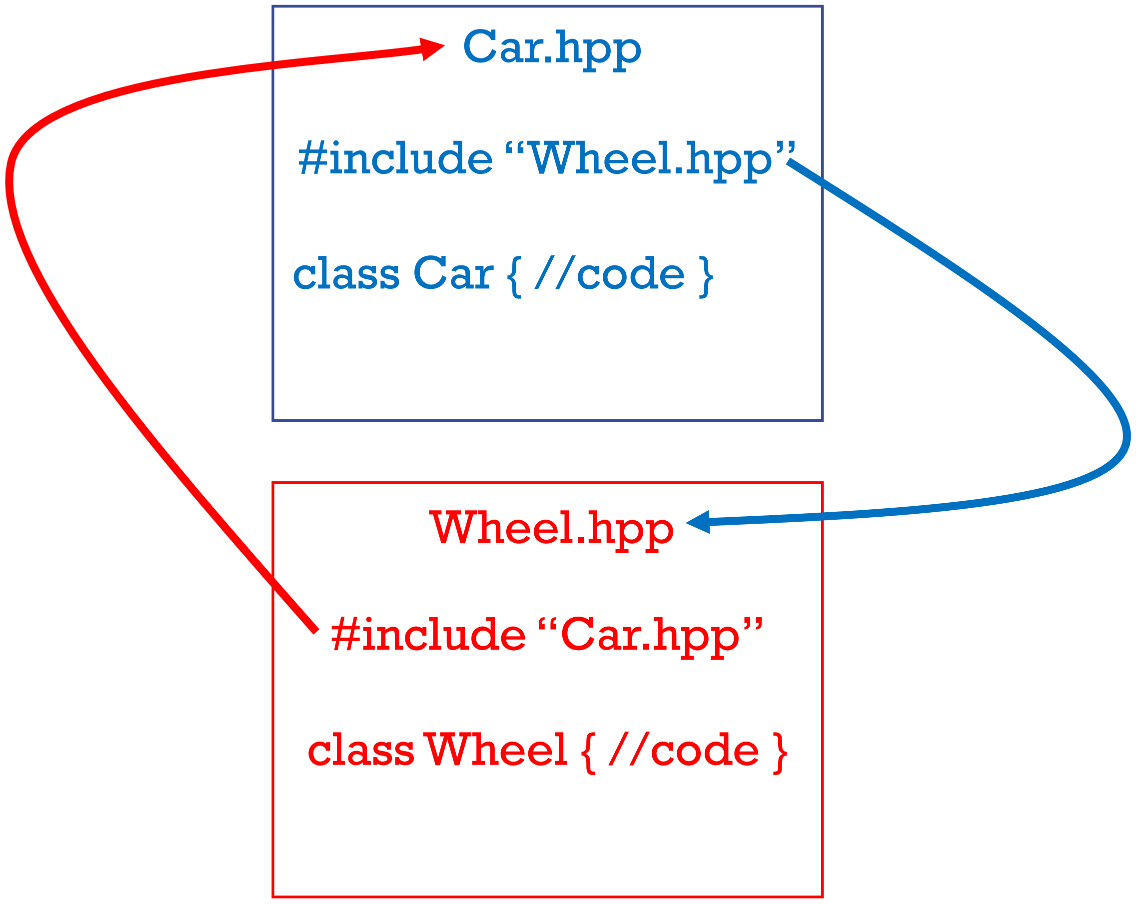
Forward declaration (motivation) (2a of 3)

```
File Car.hpp

#include "Wheel.hpp"

class Car
{
    Wheel wheels [];
}
```



Forward declaration (motivation) (2b of 3)

```
File Wheel.hpp

#include "Car.hpp" // UH OH!  THIS IS TROUBLE!

class Wheel
{
     Car * owner;
}
```



Forward declaration (motivation) (3 of 3)

- How do we include the Car inside the Wheel header file?

- If we #include "Car.hpp", then we would have to insert the Car.hpp file which includes the Wheel.hpp file which includes the Car.hpp file which includes the Wheel.hpp file…
- The compiler error message is **not helpful**
- The solution is **forward declaration**

Solution: forward declaration!

File Wheel.hpp

```
class Car; // Forward declaration (so simple!)

class Wheel
{
    Car *owner; // Must be pointer or reference
}
```

Caution

- **If you use forward declaration, you can only declare a reference or a pointer to that type**
  - Compiler does not know how to allocate object
  - If you forget this, your code will not compile and you will see a **field 'class' has incomplete type error**.

INHERITANCE

Inheritance

C++ implements everything we've seen in Java:

1. Inheritance
2. Polymorphism
3. Abstract classes and interfaces.

In C++ we talk about:

1. Base class
2. Derived class

Inheritance relationship

Java example:

class Shape { … }

class Circle **extends** Shape { … }

C++ example:

class Shape { … }

class Circle **: public** Shape { … }

Concept is the same

- Push common attributes as high into the inheritance hierarchy as possible
- Derived classes inherit all the accessible members of the base class
- Public access specifier may be replaced by private or protected in the derived class header
- This limits the most accessible level for the members inherited from the base class

Access modifiers

Access	Public	Protected	Private
Members of the same class	yes	yes	yes
Members of a derived class	yes	yes	no
Not members	yes	no	no

What is inherited in C++?

- A publically derived class inherits access to everything **except**:
  1. Constructors *
  2. Destructor *
  3. Friends
  4. Private members.

* Not inherited per such, but they are automatically called by constructors and destructor of derived class

Which base class constructor gets called?

- We can **specify** which one to call in the derived class constructor (just like Java!).
- **In C++ the call to super looks like a member initialization list.**
- Pass the parameters to the base class constructor
- If we don't, the default constructor is called (just like Java!).

- Code Examples: whichconstructor.cpp and private.cpp

POLYMORPHISM AND VIRTUAL FUNCTIONS

What about pol·y·mor·phism

/ˌpäləˈmôrfizəm/

Noun

- from the Greek roots "poly" (many) and "morphe" (form, shape, structure)
- the condition of occurring in several different forms
- a feature of a programming language that allows routines to use variables of different types at different times.

What about polymorphism?

It's EASY! (I promise!)

A pointer to a derived class is type-compatible with a pointer to its base class.

This is just like Java (remember everything is a pointer in Java).

Code Example: polymorphism.cpp

But that area member function…

- There was no area member function in Shape
- Could not use a Shape pointer to ask a Rectangle or Triangle to generate the area

Q: How can we overcome this in C++?

A: Virtual members!

Virtual member

- A base class member function that can be redefined (Java: overridden) in the derived class

- Add the **virtual** keyword to the function declaration

- **Remember: non-virtual members of the derived class <u>cannot</u> be accessed through a reference of the base class**

Virtual member

- Permits a member of the derived class with the same name as the member in the base class to be appropriately called from a pointer
- A class that declares or inherits a virtual function is called a **polymorphic class**
- Permits dynamic binding aka late binding aka polymorphic method dispatch

Code Example: virtual.cpp

More about virtual functions

- Virtual specifies that a non-static member function supports dynamic binding
- Used with pointers and references
- A call to an overridden virtual function invokes the behavior in the derived class
- We can invoke the original function by using the base class name and the scope operator (qualified name lookup)

Code Example: virtual2.cpp