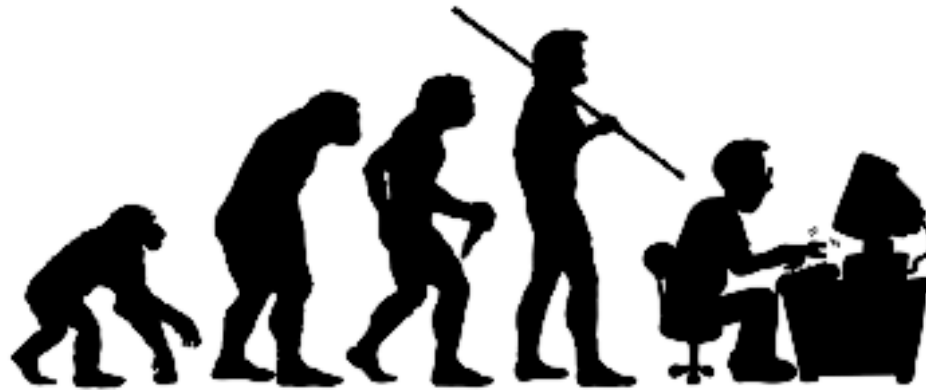


Lecture #6

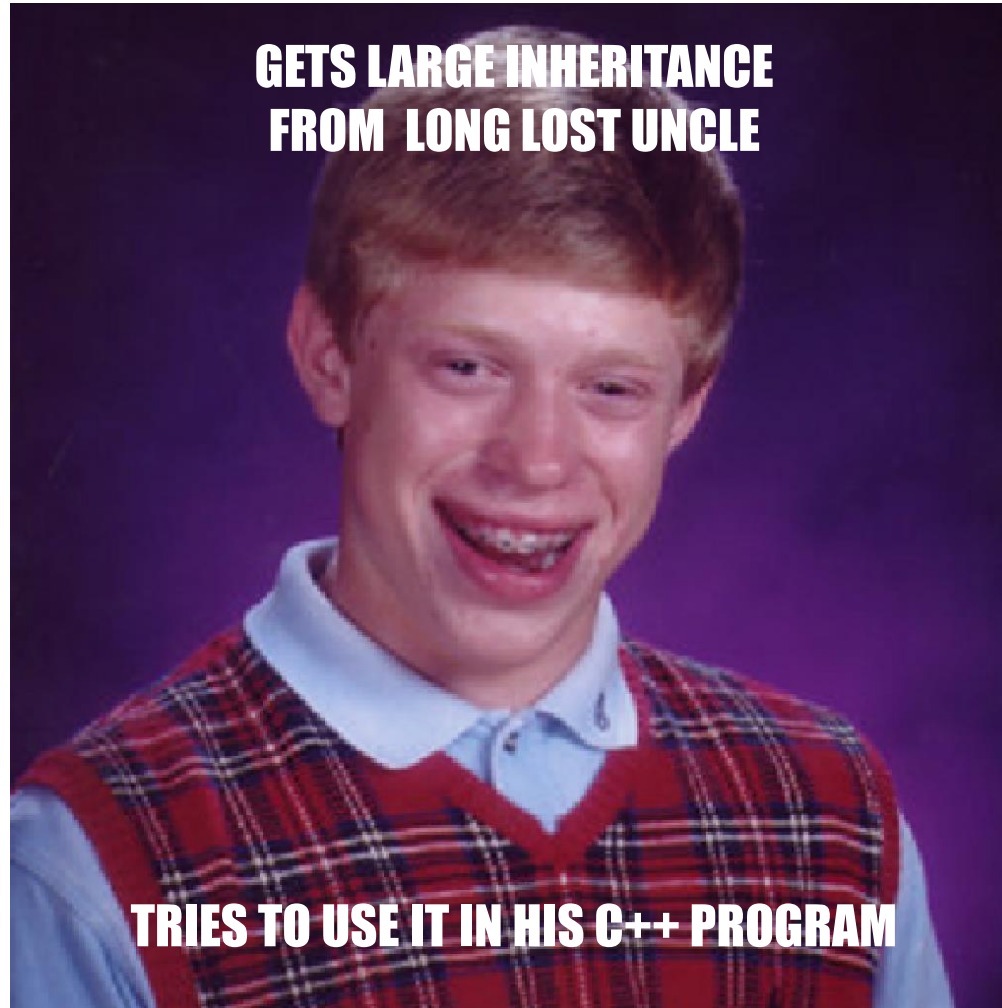
Inheritance



From Wikipedia:

"Inheritance is a way to form new classes (instances of which are called objects) using classes that have already been defined."

Inheritance



Inheritance

Why should you care?

Inheritance is the basis of all Object Oriented Programming.

Using it can dramatically **simplify your programs** and make them **more maintainable**.

And you'll almost certainly get grilled on it during internship interviews.

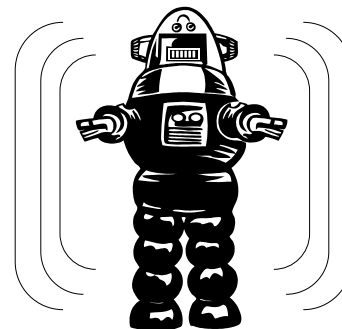
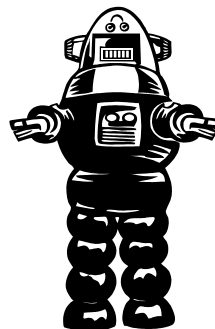
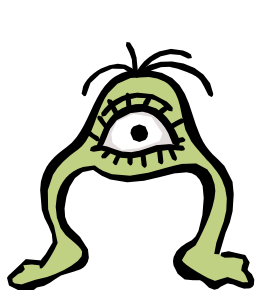
So pay attention!



Inheritance

Let's say we're writing a video game.

In the game, the player has to fight various **monsters** to save the world.

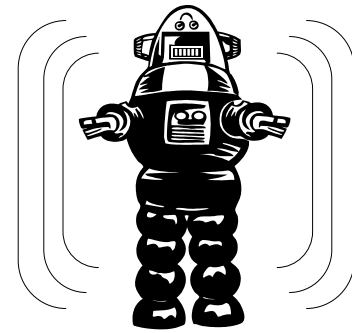


For each monster you could provide a **class definition**.

For example, consider the **Robot** class...

```
class Robot
{
public:
    void setX(int newX);
    int getX();
    void setY(int newY);
    int getY();
private:
    int m_x, m_y;
};
```

Inheritance



Now lets consider a
Shielded Robot class...

```
class Robot
{
public:
    void setX(int newX);
    int getX();
    void setY(int newY);
    int getY();
private:
    int m_x, m_y;
};
```

```
class ShieldedRobot
{
public:
    void setX(int newX);
    int getX();
    void setY(int newY);
    int getY();
    int getShield();
    void setShield(int s);
private:
    int m_x, m_y, m_shield;
};
```

Let's compare both classes...
What are their similarities?

- Both classes have **x** and **y** coordinates
- In the *Robot* class, **x** and **y** describe the position of the robot
- In the *ShieldedRobot* class **x** and **y** also describe the robot's position
- So **x** and **y** have the same purpose/meaning in both classes!
- Both classes also provide the same set of methods to **get** and **set** the values of **x** and **y**

Inheritance

```
class Robot
{
public:
    void setX(int newX);
    int getX();
    void setY(int newY);
    int getY();
private:
    int m_x, m_y;
};
```

```
class ShieldedRobot
{
public:
    void setX(int newX);
    int getX();
    void setY(int newY);
    int getY();
    int getShield();
    void setShield(int s);
private:
    int m_x, m_y, m_shield;
};
```

In fact, the only difference between a **Robot** and a **ShieldedRobot** is that a **ShieldedRobot** also has a **shield** to protect it.

A **ShieldedRobot** essentially is a kind of **Robot**!

It shares *all* of the same methods and data as a **Robot**; it just has some *additional* methods/data.

It's a pity that even though **ShieldedRobot** has just a few extra features we have to define a *whole new class* for it!

Inheritance

Here's another example...

Notice that a **Student** basically is a type of **Person**! It shares *all* of the same methods/data as a **Person** and just adds some *additional* methods/data.

```
class Person
{
public:
    string getName(void);
    void setName(string & n);
    int  getAge(void);
    void setAge(int age);
```

```
private:

    string m_sName;
    int    m_nAge;
};
```

```
class Student
{
public:
    string getName(void);
    void setName(string & n);
    int  getAge(void);
    void setAge(int age);
    int  getStudentID();
    void setStudentID();
    float getGPA();
```

```
private:
    string m_sName;
    int    m_nAge;
    int    m_nStudentID;
    float  m_fGPA;
```

Person and **Student**
are so closely
related...

Yet, to define my **Student** class, I had to write every one of its functions like **getName()**, **setAge()**, etc., from scratch!

What a waste of
time!

Inheritance

Wouldn't it be nice if C++ would let us somehow **define a new class** and have it **"inherit"** all of the methods/data of an existing, related class?

Then we wouldn't need to rewrite/copy all that code from our first class into our second class!

That's the idea behind **C++ inheritance**!

Inheritance is a technique that enables us to define a **"subclass"** (like ShieldedRobot) and have it **"inherit"** all of the functions and data of a **"superclass"** (like Robot).

Among other things, this enables you to **eliminate duplicate code**, which is a big **no-no** in software engineering!



Inheritance: How it Works

```
class Robot
{
public:
    void setX(int newX)
    { m_x = newX; }

    int getX()
    { return(m_x); }

    void setY(int newY)
    { m_y = newY; }

    int getY()
    { return(newY); }
private:
    int m_x, m_y;
};
```

Your **subclass** can now
do everything the
superclass can do,
and more!

You explicitly
tell C++ that
your **new class**
is based on an
existing class!

First you define the **superclass** and
implement all of its member functions.

Then you **define** your **subclass**,
explicitly basing it on the **superclass**...

Finally you add **new** variables and
member functions as needed.

```
class ShieldedRobot is a kind of Robot
{
public:
    // ShieldedRobot can do everything
    // a Robot can do, plus:
    int getShield()
    { return m_shield; }
    void setShield(int s)
    { m_shield = s; }
private:
    // a ShieldedRobot has x,y PLUS a
    int m_shield;
};
```

Inheritance

```
class Robot
{
public:
    void setX(int newX)
    { m_x = newX; }

    int getX()
    { return(m_x); }

    void setY(int newY)
    { m_y = newY; }

    int getY()
    { return(newY); }
private:
    int m_x, m_y;
};
```

```
class ShieldedRobot is a kind of Robot
{
public:
    // ShieldedRobot can do everything
    // a Robot does, plus:
    void setShield(int s)
    { m_shield = s; }

    int getShield()
    { return(m_shield); }

private:
    // a ShieldedRobot has x,y PLUS a
    int m_shield;
};
```

C++ automatically determines which function to call...

```
int main(void)
{
    ShieldedRobot r;
    r.setX(5);
    r.setShield(10);
    ...
}
```

r

ShieldedRobot data:
m_shield: 10

Robot data:
m_x: 5
m_y:

"Is a" vs. "Has a"

"A **Student** is a type of **Person** (plus an ID#, GPA, etc.)."

"A **ShieldedRobot** is a type of **Robot** (plus a shield strength, etc.)."

Any time we have such a relationship: "**A** is a type of **B**,"
C++ inheritance may be warranted.

```
class Person
{
public:
    string getName(void);
    void setName(string & n);
    int getAge(void);
    void setAge(int age);

private:
    string m_sName;
    int    m_nAge;
};
```

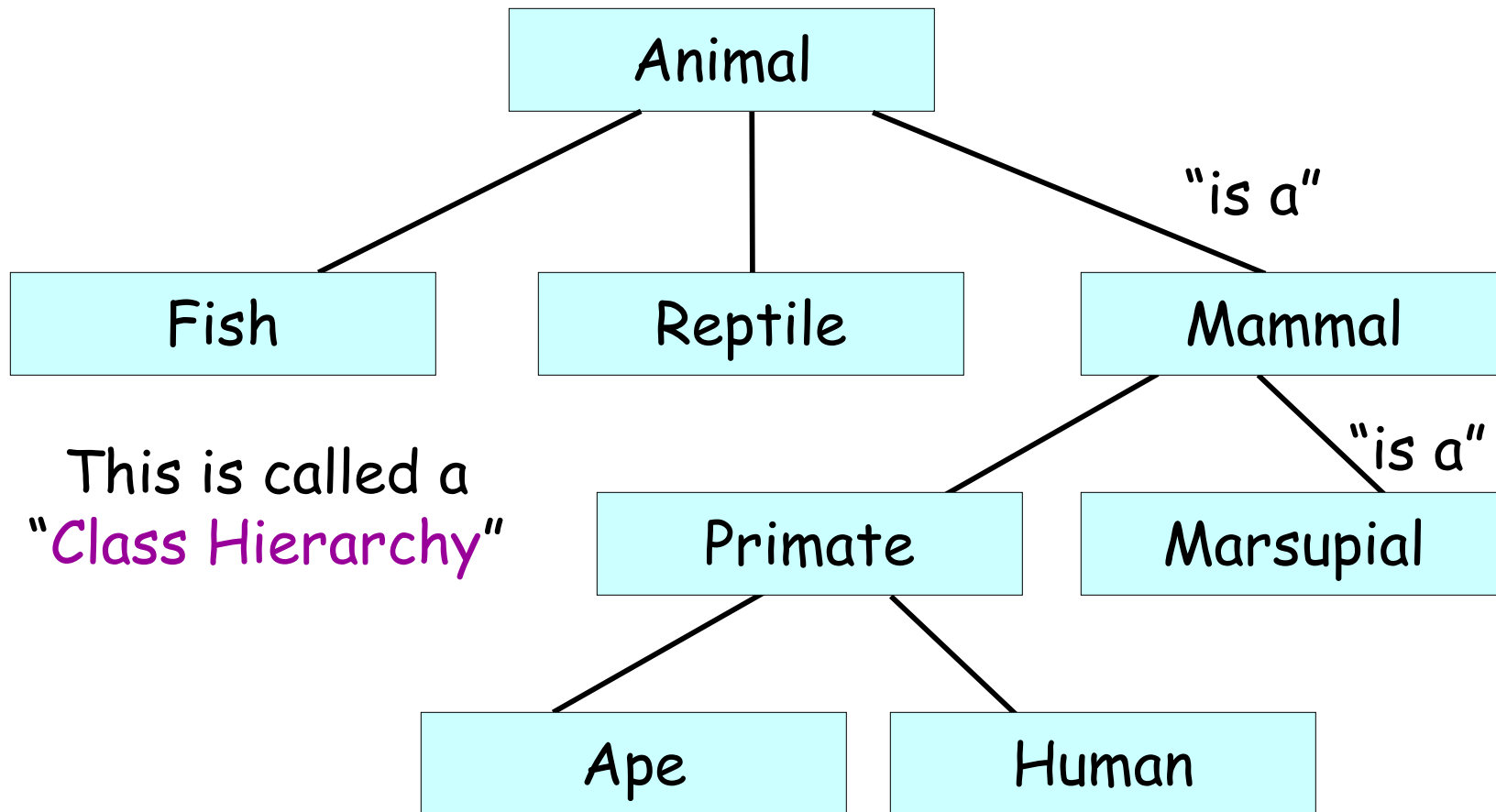
In contrast, consider a
Person and a **name**.

A person has a name,
but you **wouldn't** say that
"a **person** is a type of **name**."

In this case, you'd simply make
the name a member variable.

See the difference between
Student & Person vs. **Person & name**?

Inheritance



"A **mammal** is an **animal** (with fur)"

"A **marsupial** is a **mammal** (with a pouch)"

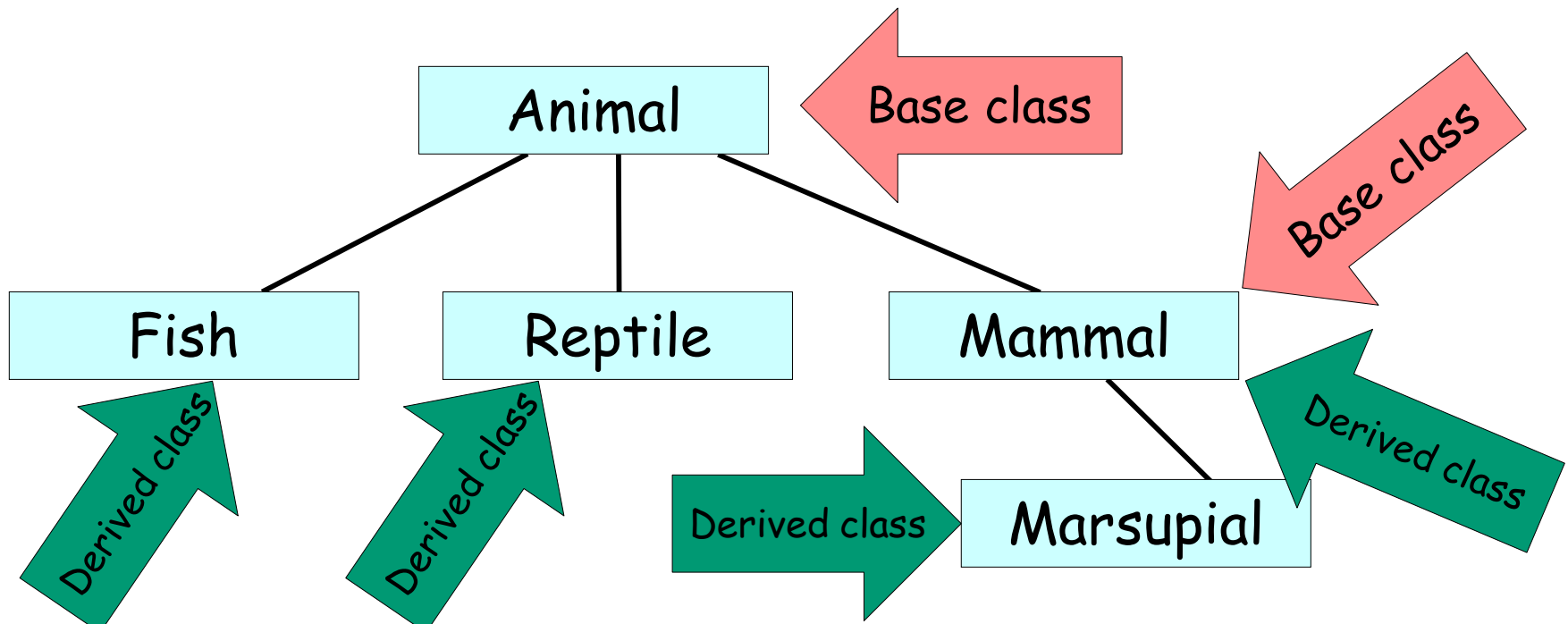
Inheritance: Terminology

A class that serves as the basis for other classes is called a **base class** or a **superclass**.

So both **Animal** and **Mammal** are **base classes**.

A class that is derived from a base class is called a **derived class** or a **subclass**.

So **Fish**, **Reptile**, **Mammal** and **Marsupial** are **derived classes**.



Inheritance

Now let's see the actual C++ syntax...

(I cheated on the previous examples.)

In C++, you can inherit more than once:

So now a CompSciStudent object can **say smart things**, has a **student ID** and she also has a **name**!

```
class Person
{
public:
    string getName(void);
    ...
private:
    string m_name;
    int m_age;
};
```

```
class Student is
    a kind of Person
{
public:
    // new stuff:
    int GetStudentID();
    ...
private:
    // new stuff:
    int m_studentID;
    ...
};
```

```
class CompSciStudent is
    a kind of Student
{
public:
    // new stuff:
    void saySomethingSmart();
private:
    // new stuff:
    string m_smartIdea;
};
```

Proper Inheritance Syntax

```
// base class
class Robot
{
public:
    void setX(int newX)
    { m_x = newX; }

    int getX()
    { return(m_x); }

    void setY(int newY)
    { m_y = newY; }

    int getY()
    { return(m_y); }

private:
    int m_x, m_y;
};
```

```
// derived class
class ShieldedRobot: public Robot
{
public:
    void setShield(int s)
    { m_shield = s; }

    int getShield()
    { return(m_shield); }

private:
    int m_shield;
};
```

This line says that **ShieldedRobot** publicly states that it is a subclass of **Robot**.

This causes our **ShieldedRobot** class to have all of the member variables and functions of **Robot** PLUS its own members as well!

The Three Uses of Inheritance



Reuse is when you **write code once** in a base class and reuse the same code in your derived classes (to reduce duplication).



Extension is when you **add new behaviors** (member functions) **or data** to a derived class that were not present in a base class.



Specialization is when you **redefine an existing behavior** (from the base class) with a new behavior (in your derived class).

Inheritance: Reuse

```
class Person
{
public:
    string getName()
    { return m_name; }
    void goToBathroom()
    { cout << "splat!"; }
    ...
};
```

```
class Whiner: public Person
{
public:
    void complain()
    {
        cout << "I hate homework!";
    }
};
```

Every **public method** in the base class is automatically reused/exposed in the derived class (just as if it were defined there).

And, as such, they may be used normally by the rest of your program.

And of course, your derived class can call them too!

```
int main()
{
    Whiner joe;

    joe.goToBathroom();
    joe.complain();
}
```

Inheritance: Reuse

```
class Person
{
public:
    string getName()
    { return m_name; }
    void goToBathroom()
    { cout << "splat!"; }
    ...
};
```

```
class Whiner: public Person
{
public:
    void complain()
    {
        cout << "I hate homework!";
    }
};
```

Every **public method** in the base class is automatically reused/exposed in the derived class (just as if it were defined there).

And, as such, they may be used normally by the rest of your program.

And of course, your derived class can call them too!

```
int main()
{
    Whiner joe;

    joe.goToBathroom();
    joe.complain();
}
```

Inheritance: Reuse

```
// base class
class Robot
{
public:
    Robot(void);
    int getX();
    int getY();
```

```
private: // methods
    void chargeBattery();
private: // data
    int m_x, m_y;
};
```

These **methods** and **variables** are **hidden** from all **derived classes** and can't be reused directly.

Only **public members** in the base class are **exposed/reused** in the **derived class(es)**!

Private members in the **base class** are **hidden** from the **derived class(es)**!

```
// derived class
class ShieldedRobot : public Robot
{
public:
    ShieldedRobot(void)
    {
        m_shield = 1;
        chargeBattery(); // FAIL!
    }
    int getShield();

private:
    int m_shield;
};
```

THIS IS ILLEGAL!

The **derived class** may **not** access **private members** of the **base class**!

Inheritance: Reuse

If you would like your **derived** class to be able to reuse one or more **private member functions** of the **base** class...

But you don't want the rest of your program to use them...

Then make them **protected** instead of **private** in the base class:

This lets your derived class (and its derived classes) reuse these member functions from the base class.

But still prevents the rest of your program from seeing/using them!

```
int main()
{
    ShieldedRobot stan;

    stan.chargeBattery(); // STILL FAILS!
}
```

```
class Robot
{
public:
    Robot(void);
    int getX() const;
    ...
protected: // methods
    void chargeBattery();
private: // data
    int m_x, m_y;
};
```

But **never ever** make your **member variables** **protected** (or **public**).

A class's member variables are for it to access alone!

If you **expose member variables** to a **derived class**, you violate encapsulation - and that's bad!

```
class ShieldedRobot : public Robot
{
public:
    ShieldedRobot(void) {
        m_shield = 1;
        chargeBattery(); // Now it's OK!
    }
    void setShield(int s);
    ...
private:
    int m_shield;
};
```

Reuse Summary

If I define a **public** member ~~variable/function~~ in a base class B:

Any function in class B may access it.

Any function in all classes derived from B may access it.

All classes/functions unrelated to B may access it.

If I define a **private** member variable/function in a base class B:

Any function in class B may access it.

No functions in classes derived from B may access it *.

No classes/functions unrelated to B may access it *.

If I define a **protected** member ~~variable/function~~ in a base class B:

Any function in class B may access it.

Any function in all classes derived from B may access it.

No classes/functions unrelated to B may access it *.

* Unless the other class/func is a "friend" of B

The Three Uses of Inheritance



Reuse is when you **write code once** in a base class and reuse the same code in your derived classes (to reduce duplication).



Extension is when you **add new behaviors** (member functions) **or data** to a derived class that were not present in a base class.



Specialization is when you **redefine an existing behavior** (from the base class) with a new behavior (in your derived class).

Inheritance: Extension

```
class Person
{
public:
    string getName()
    { return m_name; }
    void goToBathroom()
    {
        if (iAmConstipated)
            complain(); // ERROR;
    }
};
```

```
class Whiner: public Person
{
public:
    void complain()
    {
        cout << "I hate " <<
            whatIHate;
    }
private:
    string whatIHate;
};
```

Extension is the process of adding new methods or data to a derived class.

All **public extensions** may be used normally by the rest of your program.

But while these extend your derived class, they're **unknown to your base class!**

Your **base class** only **knows about itself** - it knows nothing about classes derived from it!

```
int main()
{
    Whiner joe;
    joe.complain();
}
```

The Three Uses of Inheritance



Reuse is when you **write code once** in a base class and reuse the same code in your derived classes (to reduce duplication).



Extension is when you **add new behaviors** (member functions) **or data** to a derived class that were not present in a base class.



Specialization is when you **redefine an existing behavior** (from the base class) with a new behavior (in your derived class).

Inheritance: Specialization/Overriding

In addition to **adding entirely new functions** and variables to a derived class...

You can also **override or specialize existing functions** from the base class in your derived class.

If you do this, you should **always** insert the **virtual** keyword in front of **both** the original and replacement functions!

```
class Student
{
public:
    virtual void WhatDoISay()
    {
        cout << "Go bruins!";
    }
    ...
};
```

```
class NerdyStudent: public Student
{
public:
    virtual void WhatDoISay()
    {
        cout << "I love circuits!";
    }
    ...
};
```

Inheritance: Specialization/Overriding

In addition to **adding entirely new functions** and variables to a derived class...

You can also **override or specialize existing functions** from the base class in your derived class.

If you do this, you should **always** insert the **virtual** keyword in front of **both** the original and replacement functions!

Go bruins!
I love circuits!

```
int main()
{
    Student      carey;
    NerdyStudent davidS;

    carey.WhatDoISay();
    davidS.WhatDoISay();
    ...
}
```

```
class Student
{
public:
    virtual void WhatDoISay()
    {
        cout << "Go bruins!";
    }
    ...
};
```

```
class NerdyStudent: public Student
{
public:
    virtual void WhatDoISay()
    {
        cout << "I love circuits!";
    }
    ...
};
```

Inheritance: Specialization/Overriding

If you define your member functions OUTSIDE your class, you must only use the **virtual** keyword within your **class definition**:

```
class Student
{
public:
    virtual void WhatDoISay() ;
    ...
};

void Student::WhatDoISay()
{
    cout << "Hello!";
}
```

Use **virtual** here within your class definition:

```
class NerdyStudent: public Student
{
public:
    virtual void WhatDoISay() ;
    ..
};

void NerdyStudent::WhatDoISay()
{
    cout << "I love circuits!";
}
```

Don't write **virtual** here:

Specialization: When to Use Virtual

Since the meaning of `getX()` is the same across all Robots... We will never need to redefine it... So we **won't** make it a **virtual** function.

But since subclasses of our Robot might say different things than our base Robot... We should make `talk()` **virtual** so it can be redefined!

```
class Robot
{
public:
    int getX() { return m_x; }
    int getY() { return m_y; }
    virtual void talk()
        { cout << "Buzz. Click. Beep."; }
private:
    int m_x, m_y;
};
```

You only want to use the **virtual** keyword for functions you intend to override in your subclasses.

```
class ComedianRobot: public Robot
{
public:
    // inherits getX() and getY()
    virtual void talk()
    {
        cout << "Two robots walk into a bar...";
    }
private:
    ...
};
```

Our derived class will simply inherit the original versions of `getX()` and `getY()`

Since `talk()` is **virtual** in our base class, we can safely define a new version in our derived class!

Specialization: Method Visibility

```
class Student
{
public:
    virtual void cheer()
    { cout << "go bruins!"; }
    void goToBathroom()
    { cout << "splat!"; }
    ...
};
```

```
class NerdyStudent: public Student
{
public:
    virtual void cheer()
    {
        cout << "go algorithms!";
    }
    ...
};
```

If you **redefine a function**
in the derived class...

then the **redefined version** **hides** the
base version of the function...

(But only when using your derived class)

go algorithms!
go bruins!

```
int main()
{
    NerdyStudent lily;

    lily.cheer();
}
```

```
int main()
{
    Student george;

    george.cheer();
}
```

Specialization: Reuse of Hidden Base-class Methods

```
class Student
{
public:
    virtual void cheer()
    { cout << "go bruins!"; }
    void goToBathroom()
    { cout << "splat!"; }
    ...
};
```

Your **derived class** will, by default, always use the **most derived version** of a specialized method.

If you want to call the **base class's version** of a method that's **been redefined** in the derived class...

You can do so by using the **baseclass::method()** syntax...

You can also use this syntax, although it's pretty rare.

```
class NerdyStudent: public Student
{
public:
    virtual void cheer()
    {
        cout << "go algorithms!";
    }
    void getExcitedAboutCS()
    {
        Student::cheer();
    }
};
```

C++: Ahh, since the programmer prefixed this with **Student::** I'll call Student's version of the **cheer()** function!

```
int main()
{
    NerdyStudent lily;

    lily.getExcitedAboutCS();
    lily.Student::cheer();
}
```

Specialization: Reuse of Hidden Base-class Methods

Here's how we do it!
First, you call the base-version of the method...

```
class Student
{
public:
    Student()
    {
        myFavorite = "alcohol";
    }

    virtual string whatILike()
    {
        return myFavorite;
    }

private:
    string myFavorite;
};
```

This method here...

*Needs to use this
one that it overrides.*

```
class NerdyStudent: public Student
{
public:
    virtual string whatILike()
    {
        string fav =
            Student::whatILike();

        fav += " bunsen burners"
        return fav;
    }
};
```

Then you modify any
result you get back,
as required... and
return it.

Sometimes a method in your derived class will want to rely upon the overridden version in the base class...

Let's see how this works!

```
int main()
{
    NerdyStudent carey;

    string x = carey.whatILike();
    cout << "Carey likes " << x;
}
```

Inheritance & Construction

Ok, how are super-classes and sub-classes constructed?

Let's see!



Inheritance & Constructors

```
// superclass
```

```
class Robot
```

```
{
```

```
public:
```

```
    Robot(void)
```

```
    {  
        Call m_bat's constructor
```

```
    {
```

```
        m_x = m_y = 0;
```

```
    }
```

```
    ...
```

```
private:
```

```
    int        m_x, m_y;
```

```
    Battery    m_bat;
```

```
};
```

And if you don't explicitly construct
your member variables (objects),
C++ does it for you!

Before C++ can run
your constructor body...

It must first construct
its member variables (objs)!

Forget about **inheritance** for a second and think back a few
weeks to **class construction**...

So we know that C++ automatically constructs an object's member
variables first, then runs the object's constructor...

Inheritance & Construction

```
// superclass
class Robot
{
public:
    Robot(void)
        Call m_bat's constructor
    {
        m_x = m_y = 0;
    }
    ...
private:
    int      m_x, m_y;
    Battery  m_bat;
};
```

```
// subclass
class ShieldedRobot: public Robot
{
public:
    ShieldedRobot(void)
        Call m_sg's constructor
    {
        m_shieldStrength = 1;
    }
    ...
private:
    int m_sheildStrength;
    ShieldGenerator m_sg;
};
```

And as you'd guess, C++ also does this for derived classes...

Inheritance & Construction

```
// superclass
class Robot
{
public:
    Robot(void)
        Call m_bat's constructor
    {
        m_x = m_y = 0;
    }
    ...
private:
    int      m_x, m_y;
    Battery  m_bat;
};
```

```
// subclass
class ShieldedRobot: public Robot
{
public:
    ShieldedRobot(void)
        Call m_sg's constructor
    {
        m_shieldStrength = 1;
    }
    ...
private:
    int m_sheildStrength;
    ShieldGenerator m_sg;
};
```

But when you define a derived object, it has both superclass and subclass parts...

And both need to be constructed!

So which one is constructed first?

```
int main()
{
    ShieldedRobot phyllis;
}
```

Inh

& Const

It also does the same thing to initialize the base part of the object!

Just as C++ added an implicit call to initialize ShieldedRobot's member variables...

```
// superclass
```

```
class Robot
```

```
{
```

```
public:
```

```
Robot(void)
```

Call m_bat's constructor

```
{
```

```
m_x = m_y = 0;
```

```
}
```

```
// subclass
```

```
class ShieldedRobot
```

```
{
```

```
public:
```

```
ShieldedRobot(void)
```

Call m_sg's constructor

```
{
```

```
m_shieldStrength = 1;
```

phyllis

Robot's data:

m_x m_y
m_bat

ShieldedRobot's data:

m_shieldStrength
m_sg

C++ runs the base class's constructor first.

Then it runs the derived class's constructor after.

Answer: C++ always constructs the base part first, then the derived part second!

And it does this by secretly modifying your derived constructor - just as it did to construct your member variables!

```
int main()
```

```
{
```

```
ShieldedRobot phyllis;
```

```
}
```

Inheritance & Construction

It also does the same thing to initialize the base part of the object!

```
// superclass
class Robot
{
public:
```

```
    Robot(void)
```

Call m_bat's constructor

```
{
    m_x = m_y = 0;
}
```

phyllis

Robot's data:

m_x m_y
m_bat

ShieldedRobot's data:

m_shieldStrength
m_sg

```
// subclass
```

```
class ShieldedRobot: public Robot
{
public:
```

```
    ShieldedRobot(void)
```

Call Robot's constructor

Call m_sg's constructor

```
{
    m_shieldStrength = 1;
}
...
```

```
private:
```

```
    int m_sheildStrength;
    ShieldGenerator m_sg;
};
```

Answer: C++ always constructs the basic part first, then the derived part second!

And it does this by secretly modifying your derived constructor - just as it did to construct your member variables!

```
int main()
```

```
{
    ShieldedRobot phyllis;
}
```

Inheritance & Construction

```
// superclass
```

```
class Robot
```

```
{
```

```
public:
```

```
    Robot(void)
```

Call m_bat's constructor

```
{
```

```
    m_x = m_y = 0;
```

```
}
```

```
// subclass
```

```
class ShieldedRobot: public Robot
```

```
{
```

```
public:
```

Call Robot's constructor

Call m_sg's constructor

```
{
```

```
    m_shieldStrength = 1;
```

```
private:
```

```
    int m_sheildStrength;
```

```
    ShieldGenerator m_sg;
```

```
};
```

phyllis

Robot's data:

m_x m_y

m_bat

ShieldedRobot:

m_shieldStrength

m_sg

Finally, C++ runs the body of the derived c'tor!

So any time you define a derived object...

C++ first (implicitly) calls your base c'tor...

Then C++ (implicitly) constructs your derived object's member variables...

Last, C++ runs the body of your derived c'tor!

```
int main()
```

```
{
```

```
    ShieldedRobot phyllis;
```

```
}
```

ce & Co

```
class Battery
{
public:
    Battery()
    { ... }
};
```

```
public:
    Robot(void)
    {
        Call m_bat's constructor
        {
            m_x = m_y = 0;
        }
    }
```

phyllis

Robot's data:

m_x	0	m_y	0
m_bat	Full		

ShieldedRobot's data:

m_shieldStrength	1
m_sg	On

```
class ShieldGenerator
{
public:
    ShieldGenerator()
    { ... }
};
```

```
// subcl
class Sh
{
public:
    ShieldedRobot(void)
    {
        Call Robot's constructor
        Call m_sg's constructor
        {
            m_shieldStrength = 1;
        }
        ...
    }

private:
    int m_sheildStrength;
    ShieldGenerator m_sg;
};
```

Alright, let's see the whole thing in action!

```
int main()
{
    ShieldedRobot phyllis;
}
```

Inheritance

Construction

```
class Machine
{
public:
    Machine()
    { #3 }
};
```

```
// superclass
class Robot: public Machine
{
public:
    Robot(void)
    {
        Call Machine's constructor #2
        Call m_bat's constructor #4
        {
            m_x = m_y = 0; #5
        }
        ...
private:
    int      m_x, m_y;
    Battery  m_bat;
};
```

```
// subclass
class ShieldedRobot: public Robot
{
public:
    ShieldedRobot(void)
    {
        Call Robot's constructor #1
        Call m_sg's constructor #6
        {
            m_shieldStrength = 1; #7
        }
        ...
private:
    int m_sheildStrength;
    ShieldGenerator m_sg;
};
```

And of course, this applies if you inherit more than one time!

Inheritance & Destruction



Inheritance & Destruction

// superclass

class Robot

{

public:

~Robot()

{

m_bat.discharge();

}

Call m_bat's destructor

private:

int m_x, m_y;

Battery m_bat;

};

First C++ runs the body of
your outer object's d'tor...

// subclass

class ShieldedRobot: public Robot

{

public:

~ShieldedRobot()

{

m_sg.turnGeneratorOff();

}

Call m_sg's destructor

...

private:

int m_shieldStrength;

ShieldGenerator m_sg;

Then C++ destructs **all**
member objects.

OK, so how does destruction work with inheritance?

Remember that C++ implicitly destructs *all* of an object's member variables after the outer object's destructor runs.

And of course, this applies for derived objects too!

Inheritance & Destruction

```
// superclass
class Robot
{
public:
    ~Robot()
    {
        m_bat.discharge();
    }
    ...
private:
    int      m_x, m_y;
    Battery  m_bat;
};
```

Call m_bat's destructor

```
// subclass
class ShieldedRobot: public Robot
{
public:
    ~ShieldedRobot()
    {
        m_sg.turnGeneratorOff();
    }
    ...
private:
    int m_sheildStrength;
    ShieldGenerator m_sg;
};
```

Call m_sg's destructor

But when you define a derived object, it has both superclass and subclass parts...

And both need to be destructed!

So which one is destructed first?

```
int main()
{
    ShieldedRobot phyllis;
    ...
} // phyllis is destructed
```

Inheritance & Destruction

```
// superclass
class Robot
{
public:
    ~Robot()
    {
        m_bat.discharge();
    }
}
```

Call m_bat's destructor

```
// subclass
class ShieldedRobot: public Robot
{
public:
    ~ShieldedRobot()
    {
        m_sg.turnGeneratorOff();
    }
}
```

destructor

And then the base part second.

C++ destructs the derived part first...

phyllis

Robot's data:

m_x 0 m_y 0
m_bat Full

ShieldedRobot's data:

m_shieldStrength 1
m_sg On

Answer: C++ destructs the derived part first, then the base part second.

And it does this by secretly modifying your derived destructor - just as it did to destruct your member variables!

```
int main()
{
    ShieldedRobot phyllis;
    ...
} // phyllis is destroyed
```

Inheritance & Destruction

```
// superclass
```

```
class Robot
```

```
{
```

```
public:
```

```
~Robot()
```

```
{
```

```
    m_bat.discharge();
```

```
}
```

Call m_bat

```
private:
```

```
int
```

```
Batter
```

```
};
```

Robot's data:

m_x 0 m_y 0

m_bat Full

ShieldedRobot's data:

m_shieldStrength

m_sg

First C++ runs the body of your derived destructor.

Then C++ destructs all member objects in the derived part.

Finally, C++ asks the base object to destruct itself in the same manner.

```
// subclass
```

```
class ShieldedRobot: public Robot
```

```
{
```

```
public:
```

```
~ShieldedRobot()
```

```
{
```

```
    m_sg.turnGeneratorOff();
```

```
}
```

Call m_sg's destructor

Call Robot's destructor

```
private:
```

```
int m_shieldStrength;
```

```
ShieldGenerator m_sg;
```

```
int main()
```

```
{
```

```
    ShieldedRobot phyllis;
```

```
    ...
```

```
} // phyllis is destructed
```

Answer: C++ destructs the derived part first, then the base part second.

And it does this by secretly modifying your derived destructor - just as it did to destruct your member variables!

ce & De

```
class Battery
{
public:
    ~Battery()
    { ... }
};
```

```
public:
    ~Robot()
    {
        m_bat.discharge();
    }
```

Call m_bat's destructor

```
...
private:
    int
    Battery
};
```

Robot's data:

m_x	0
m_y	0
m_bat	Full

ShieldedRobot's data:

m_shieldStrength	1
m_sg	On

```
class ShieldGenerator
{
public:
    ~ShieldGenerator()
    { ... }
};
```

```
// subclass
class ShieldedRobot
{
public:
    ~ShieldedRobot()
    {
        m_sg.turnGeneratorOff();
    }
```

Call m_sg's destructor

Call Robot's destructor

```
...
private:
    int m_shieldStrength;
    ShieldGenerator m_sg;
};
```

Alright, let's see the whole thing in action!

```
int main()
{
    ShieldedRobot phyllis;
    ...
} // phyllis is destructed
```

Inheritance

```
class Machine
{
public:
    ~Machine()
    { #7 }
};
```

```
// superclass
class Robot: public Machine
{
public:
    ~Robot()
    {
        m_bat.discharge(); #4
    }
    Call m_bat's destructor #5
    Call Machine's destructor #6
private:
    int      m_x, m_y;
    Battery  m_bat;
};
```

```
// subclass
class ShieldedRobot: public Robot
{
public:
    ~ShieldedRobot()
    {
        m_sg.turnGeneratorOff(); #1
    }
    Call m_sg's destructor #2
    Call Robot's destructor #3
    ...
private:
    int m_sheildStrength;
    ShieldGenerator m_sg;
};
```

And of course, this applies if you inherit more than one time!

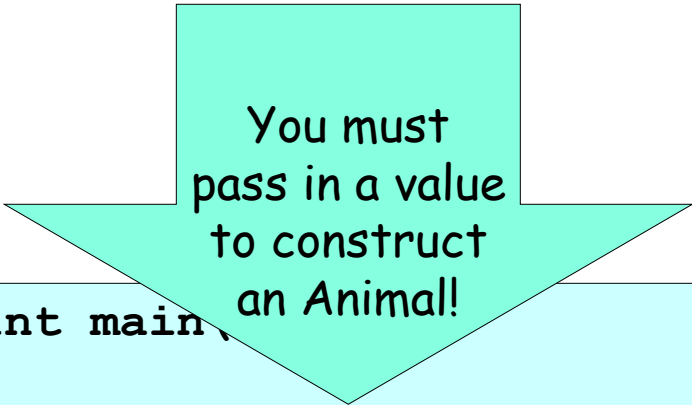
Inheritance & Initializer Lists

Consider the following
base class: **Animal**

```
class Animal
{
public:
    Animal(int lbs)
        {m_lbs = lbs;}

    void what_do_i_weigh()
        {cout << m_lbs << "lbs!\n"; }

private:
    int      m_lbs;
};
```



You must
pass in a value
to construct
an Animal!

```
int main()
{
    Animal  a(10);    // 10 lbs

    a.what_do_i_weigh();
}
```

When you construct an **Animal**,
you must specify the animal's weight.

Inheritance &_INITIALIZER Lists

Now consider the **Duck** class.
It's a subclass of **Animal**.

This c'tor
requires

a parameter!

```
class Animal
{
public:
    Animal(int lbs)
        {m_lbs = lbs;}

    void what_do_i_weigh(void)
        {cout << m_lbs << "lbs!\n"; }

private:
    int    m_lbs;
};
```

Doesn't
pass any
parameters
in!

```
class Duck : public Animal
{
public:
    Duck()
        { Call Animal() constructor
          { m_feathers = 99; }

    void who_am_i()
        { cout << "A duck!"; }

private:
    int m_feathers;
};
```

We have a **problem**! Can anyone see what it is?

Right! Our Animal constructor **requires** a **parameter**...

But our Duck class uses C++'s implicit construction mechanism...

And it doesn't pass any parameters in!

Inheritance & Initializer Lists

So what can we do?

If this
c'tor
requires
parameters!

```
class
{
public:
    Animal(int lbs)
    {m_lbs = lbs;}

    void what_do_i_weigh(void)
    {cout << m_lbs << "lbs!\n"; }

private:
    int    m_lbs;
};
```

```
class Duck : public Animal
{
public:
    Duck() : Animal(2)
    { m_feathers = 9; }

    void who_weighs(void)
    { cout << "Duck weighs " << m_lbs << "lbs!\n"; }

private:
    int m_feat
};
```

This states that before we can construct a Duck, we must first construct the Animal base part of our object!

And in this case all Ducks would weigh 2 pounds.

Then you must use an initializer list here!

Rule: If a superclass requires parameters for construction, then you **must** add an initializer list to the subclass constructor!

The first item in your initializer list must be...

the name of the base class, along with parameters in parentheses.

Of course, then C++ **doesn't implicitly call** the **base's c'tor** anymore!

Inheritance & Init

```
class Animal
{
public:
    Animal(int lbs)
        {m_lbs = lbs;}

    void what_do_i_weigh(void)
        {cout << m_lbs << "lbs!\n"; }

private:
    int    m_lbs;
};
```

```
class Stomach
{
public:
    Stomach(int howMuchGas)
        { ... }
};
```

```
class Duck : public Animal
{
public:
    Duck() : Animal(2), m_belly(1)
        { m_feathers = 99; }

    void who_am_i()
        { cout << "A duck!"; }

private:
    int m_feathers;
}; Stomach m_belly;
```

And if your derived class has **member objects**...
 whose **c'tors require parameters**...
 they can be **initialized in this way**...

Inheritance & Initializer Lists

```
class Animal
{
public:
    Animal(int lbs)
        {m_lbs = lbs;}

    void what_do_i_weigh(void)
        {cout << m_lbs << "lbs!\n"; }

private:
    int      m_lbs;
};
```

```
class Duck : public Animal
{
public:
    Duck() : Animal(2)
        { m_feathers = 99; }

    void who_am_i()
        { cout << "A duck!"; }

private:
    int m_feathers;
};
```

daffy

Duck data:
m_feathers: 99

Animal data:
m_lbs: 2

```
int main()
{
    Duck daffy;
    daffy.who_am_i();
    daffy.what_do_i_weigh();
}
```

Inheritance & Initializer Lists

Alright, let's change our **Duck** class so you can specify the **weight of a duck** during construction.

```
class Animal // base class
{
public:
    Animal(int lbs)
        {m_lbs = lbs;}

    void what_do_i_weigh(void)
        {cout << m_lbs << "lbs!\n"; }

private:
    int      m_lbs;
};
```

```
class Duck : public Animal
{
public:
    Duck(int lbs) : Animal(lbs)
        { m_feathers = 99; }

    void who_am_i()
        { cout << "A duck!"; }

private:
    int m_feathers;
};
```

Now, any time we construct a **Duck**, we must pass in its **weight**. This is then passed on to the **Animal**.

daffy

Duck data:
m_feathers: 99

Animal data:
m_lbs: 50

```
int main()
{
    Duck daffy(50); // fat!
    daffy.who_am_i();
    daffy.what_do_i_weigh();
}
```

Inheritance & Initializer Lists

Next, let's update the **Duck** class so it loses one pound the day it is born (constructed).

```
class Animal // base class
{
public:
    Animal(int lbs)
        {m_lbs = lbs;}

    void what_do_i_weigh(void)
        {cout << m_lbs << "lbs!\n"; }

private:
    int      m_lbs;
};
```

```
int main()
{
    Duck  daffy(13,75);
    daffy.who_am_i();
    daffy.what_do_i_weigh();
}
```

```
class Duck : public Animal
{
public:
    Duck(int lbs, int numF) :
        Animal(lbs-1)
        { m_feathers = numF; }

    void who_am_i()
        { cout << "A duck!"; }

private:
    int m_feathers;
};
```

daffy

Now let's update the **Duck** class so you can pass in the number of feathers when you construct it.

Duck data:
m_feathers: 75

Animal data:
m_lbs: 12

Inheritance & Initializer Lists

X

Mallard data:
myName: "Ed"

Duck data:
m_feathers: 50

Animal data:
m_lbs: 4

```
class Animal // base class
{
public:
    Animal(int lbs)
        {m_lbs = lbs;}

    void what_do_i_weigh(void)
        {cout << m_lbs << "lbs!\n"; }
```

```
private:
    class Duck : public Animal
    {
    public:
        Duck(int lbs, int numF) :
            Animal(lbs-1)
            { m_feathers = numF; }

        void who_am_i()
            { cout << "A duck!"; }

    private:
        int m_feathers;
    };
};
```

```
int main()
{
    Mallard x("Ed");
    x.who_am_i();
    x.what_do_i_weigh();
}
```

Finally let's define a subclass called **Mallard**:

- All Mallard ducks weigh 5 pounds, and have 50 feathers.
- You can specify the Mallard's name during construction.

```
class Mallard : public Duck
{
public:
    Mallard(string &name) :
        Duck(5, 50)
        { myName = name; }

private:
    string myName;
};
```

Inheritance & Assignment Ops

```
class Robot
{
public:
    void setX(int newX);
    int getX();
    void setY(int newY);
    int getY();
private:
    int m_x, m_y;
};
```

```
class ShieldedRobot: public Robot
{
public:
    int getShield ();
    void setShield(int s);
private:
    int m_shield;
};
```

What happens if I assign
one instance of a derived
class to another?

```
int main()
{
    ShieldedRobot larry, curly;

    larry.setShield(5);
    larry.setX(12);
    larry.setY(15);

    curly.setShield(75);
    curly.setX(7);
    curly.setY(9);

    ...
    larry = curly;    // what happens?
}
```


Inheritance & Assignment Ops

```
int main()
{
    ShieldedRobot larry, curly;
    ...
    larry = curly; // hmm?
}
```

It works fine.

C++ **first** copies the **base** data, from curly to larry, and **then** copies the **derived** data from curly to larry (using the operator=/copy c'tor, if present).

larry

ShieldedRobot data:

m_shield: 5

Robot data:

m_x: 12

m_y: 15

curly

ShieldedRobot data:

m_shield: 75

Robot data:

m_x: 7

m_y: 9

However, if your base and derived classes have dynamically allocated member variables (or would otherwise need a special copy constructor/assignment operator)...

then you **must** define assignment ops and copy c'tors for the base **class** and also special versions of these fns for the **derived** class!

```

class Person
{
public:
    Person() { myBook = new Book; }      // I allocate memory!!!
    Person(const Person &other);
    Person& operator=(const Person &other);

    ...
private:
    Book *myBook;
};

class Student: public Person
{
public:
    Student(const Student &other) : Person(other)
    {
        ... // make a copy of other's linked list of classes...
    }
    Student& operator=(const Student &other)
    {
        if (this == &other) return *this;
        Person::operator=(other);
        ... // free my classes and then allocate room for other's list of classes
        return(*this);
    }
private:
    LinkedList *myClasses;
};

```

Inheritance Review

Inheritance is a way to form new classes using classes that have already been defined.

Reuse

Reuse is when you **write code once** in a base class and reuse the same code in your derived classes (to save time).

Extension

Extension is when you **add new behaviors** (member functions) **or data** to a derived class that were not present in a base class.

Car → void **accelerate()**, void **brake()**, void **turn**(float angle)
Bat Mobile: public Car → void **shootLaser**(float angle)

Specialization

Specialization is when you **redefine an existing behavior** (from the base class) with a new behavior (in your derived class).

Car → void **accelerate()** { **addSpeed(10);** }
Bat Mobile: public Car → void **accelerate()** { **addSpeed(200);** }