OOP in C++

Object Oriented Programming

Four major features:

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

Encapsulation

Controlling access to the data of an object

- Sometimes called “Data Hiding”

Other languages

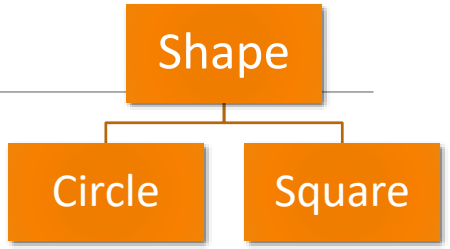
- Handled with getters & setters in Java
- Handled with Properties in C#

C++ uses getters and setters as well

Encapsulation Example

```
class Car
{
public:
	Car( );

	// Getters
	int   getMiles( );
	char* getColor( );

	// Setters – Can't directly set miles
	void setColor(char* newColor);

private:
	int   miles;
	char* color;
};
```

Inheritance



Basing a class on another class

- Parent – Child relationship
- Promotes code reuse

C# and Java support single inheritance

- Can only inherit from a single class at most

C++ supports multiple inheritance

- Can inherit from any number of classes
- Sometimes problematic

Multiple Inheritance Issues

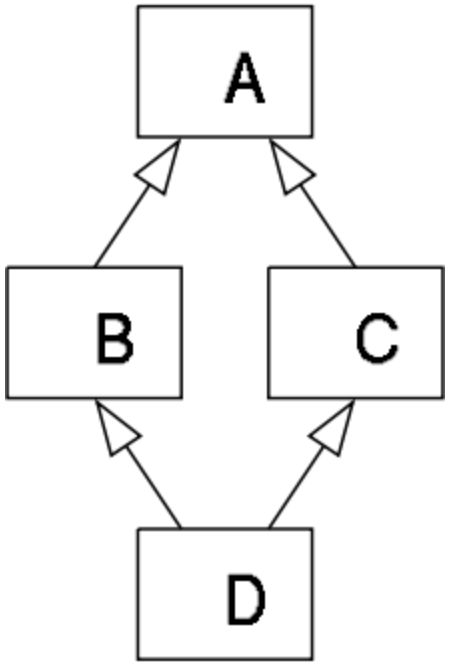
The “Diamond Problem”

A is the base (parent) class

- B and C inherit from A
- D inherits both B and C

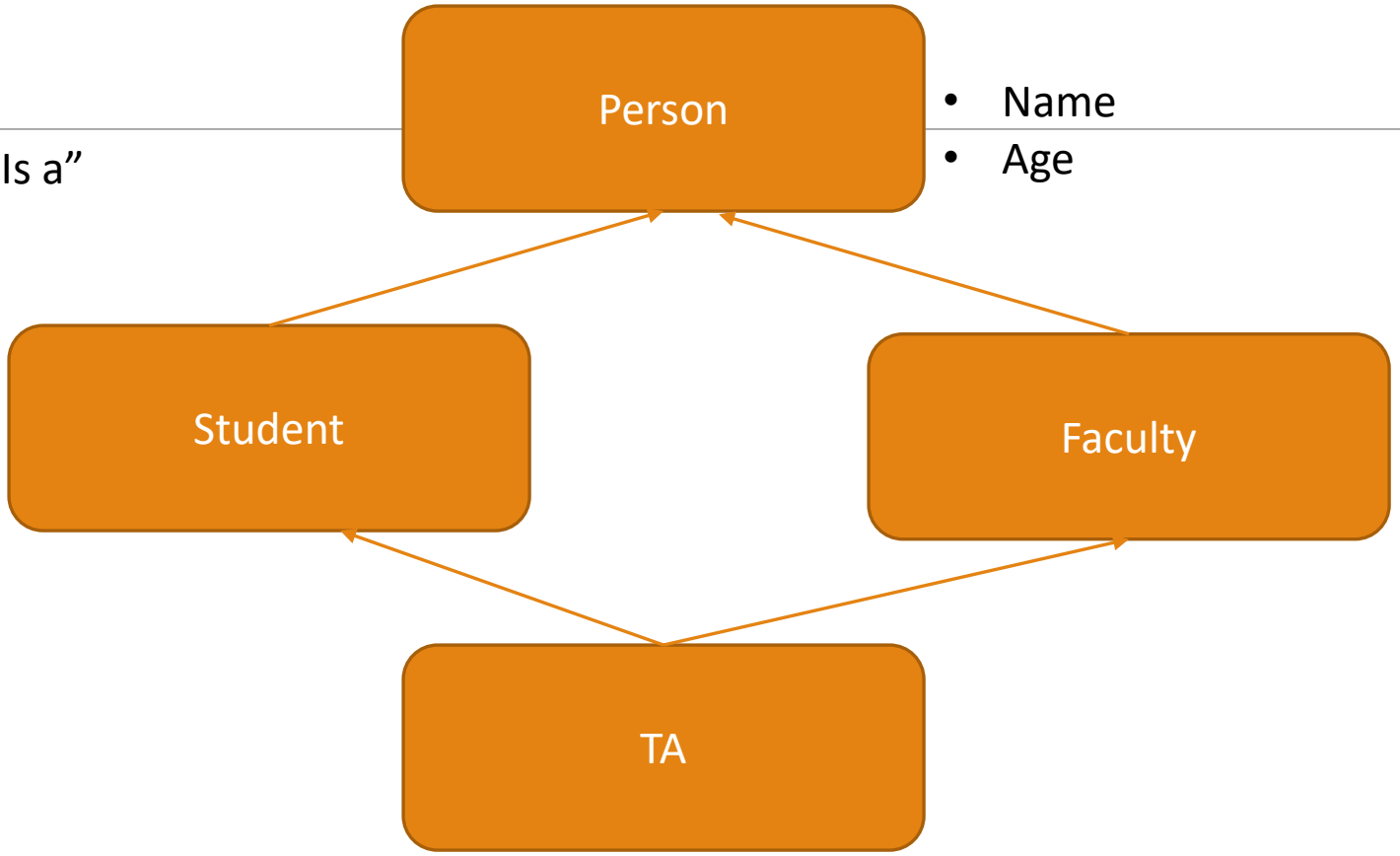
What if B and C each override a method from A?

- Which version does D get?



Example

Use the “Is a” syntax




```
class TA : public Faculty, public Student
 { …
};
```

Access Levels

Same basic access levels as other languages

Public – All code can access

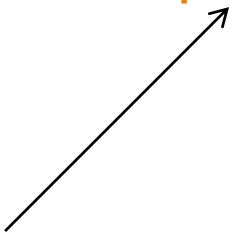
Private – Only accessible to code in this class

Protected – Accessible to code in this class
and any child classes

Inheritance Syntax

Syntax should look mostly familiar

```
class Child : public Parent
{ };
```



Notice the access specifier

- Specifies the access of the inheritance
- Which parts “outside code” has access to

Public vs. Private Inheritance

```
class Child : public Parent { };
```

From outside the Child class, the Parent's public inherited members can be accessed

This is how inheritance works in C# & Java

Public vs. Private Inheritance

```
class Child : private Parent { };
class Child : Parent { };
// private by default!
```

From outside the Child class, the Parent's public inherited members are inaccessible!

- Child can still access them, but no one else can

As if they were actually private members to begin with

Inheritance Example

```
class base

{
        public:
                int x;
        protected:
                int y;
        private:
                int z;
};
```

```
class publicDerived: public base
{
        // x is public
        // y is protected
        // z is not accessible from publicDerived
};

class protectedDerived: protected base
{
        // x is protected
        // y is protected
        // z is not accessible from protectedDerived
};

class privateDerived: private base
{
        // x is private
        // y is private
        // z is not accessible from privateDerived
}
```

Base Class Members

Access base class members by preceding the member name with the class name

```
void Child::printChild( )
{
      Parent::printParent( );
      cout << “Also a child!” << endl;
}
```

There’s no “base” keyword

Base Class Constructors

We have the following class hierarchy:

```
class Child : public Parent
{ /* code omitted */ };
```

The Child constructor will automatically call the Parent's default constructor

```
Child::Child(int a, int b)
{
    // Parent's default constructor automatically called
    // Constructor code here
}
```

Base Class Constructors

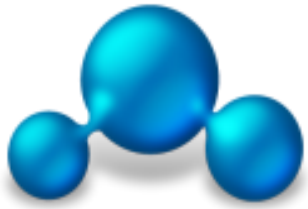
Every class in the inheritance hierarchy must have one of its constructors called

- Either implicitly - automatically
- Or explicitly – as show below

Calling Parent's constructor from Child:

```
Child::Child(int a, int b) : Parent(a, b)
{
    // Constructor code here
}
```

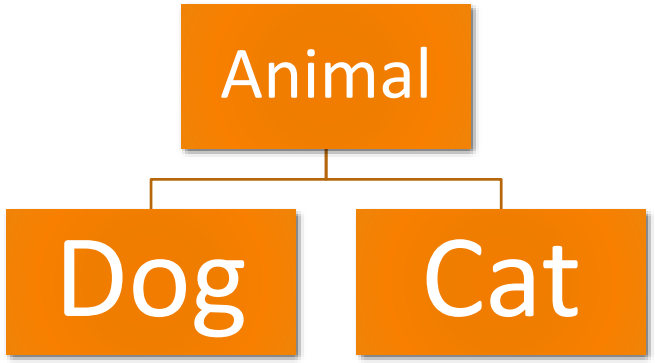
Polymorphism



Treating a child class object as an object of its parent class

Literally means “many forms”

Works with variables and pointers!



Polymorphism with Variables

```
// Create some objects (on the stack)
Dog dog = Dog( );
Cat cat = Cat( );
Animal animal = Animal( );

// Attempt to store a child in a parent variable
Animal a2 = dog;                // Implicit cast
Animal a3 = (Animal)cat;        // Explicit cast

// Cats and dogs are animals!
```

Polymorphism with Pointers

```
// Create some objects (on the heap)
Dog* dog = new Dog( );
Cat* cat = new Cat( );
Animal* animal = new Animal( );

// Attempt to store a child in a parent variable
Animal* a2 = dog;              // Implicit cast
Animal* a3 = (Animal*)cat;     // Explicit cast

// Cat pointers and dog pointers are
// animal pointers
```

Virtual methods

Virtual methods are intended to be over-ridden by derived (child) classes

E.g.

```
class Animal
{
    public:
    Animal(); //Default ctor
    virtual void Speak();//Will over-ride
…
    private:
    …
};
```

ss

```
class Horse
{
    public:
    …
    void Speak(); //Make it neigh!!
…
    private:
    …
};
```

Upcast and downcast

Downcast:

- Convert parent class to child

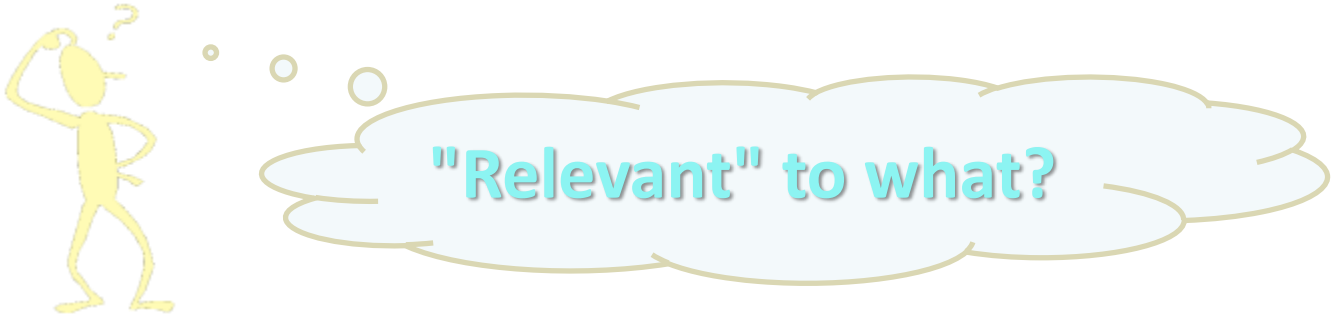
Upcast:

- Convert child class to parent

```
Parent parent;
Child child;

// upcast - implicit type cast allowed
Parent *pParent = &child;

// downcast - explicit type case required
Child *pChild =  (Child *) &parent;

pParent -> sleep();
pChild -> gotoSchool();
```

Abstraction

Abstraction means removing details of features, properties, or functions and emphasizing the more important/ relevant ones ...



... relevant to the given project (with an eye to future reuse in similar projects)

Abstraction => managing complexity

More Abstraction

Abstraction is something we do every day

- Looking at an object, we see those things about it that have meaning to us
- We abstract the properties of the object, and keep only what we need
- e.g. students get "name" but not "color of eyes"

Allows us to represent a complex reality in terms of a simplified model

Abstraction highlights the properties of an entity that we need and hides the others

Abstraction in C++

Abstraction is accomplished by hiding details of implementation

```
#include <iostream>
using namespace std;

class sample {
public:
   int g1, g2;

public:
   void val()
   {
      cout << "Enter Two values : "; cin >> gl >> g2;
   }
   void display()
   {
      cout << g1 << " " << g2;
      cout << endl;
   }
};
int main()
{
   sample S;
   S.val();
   S.display();
}
```