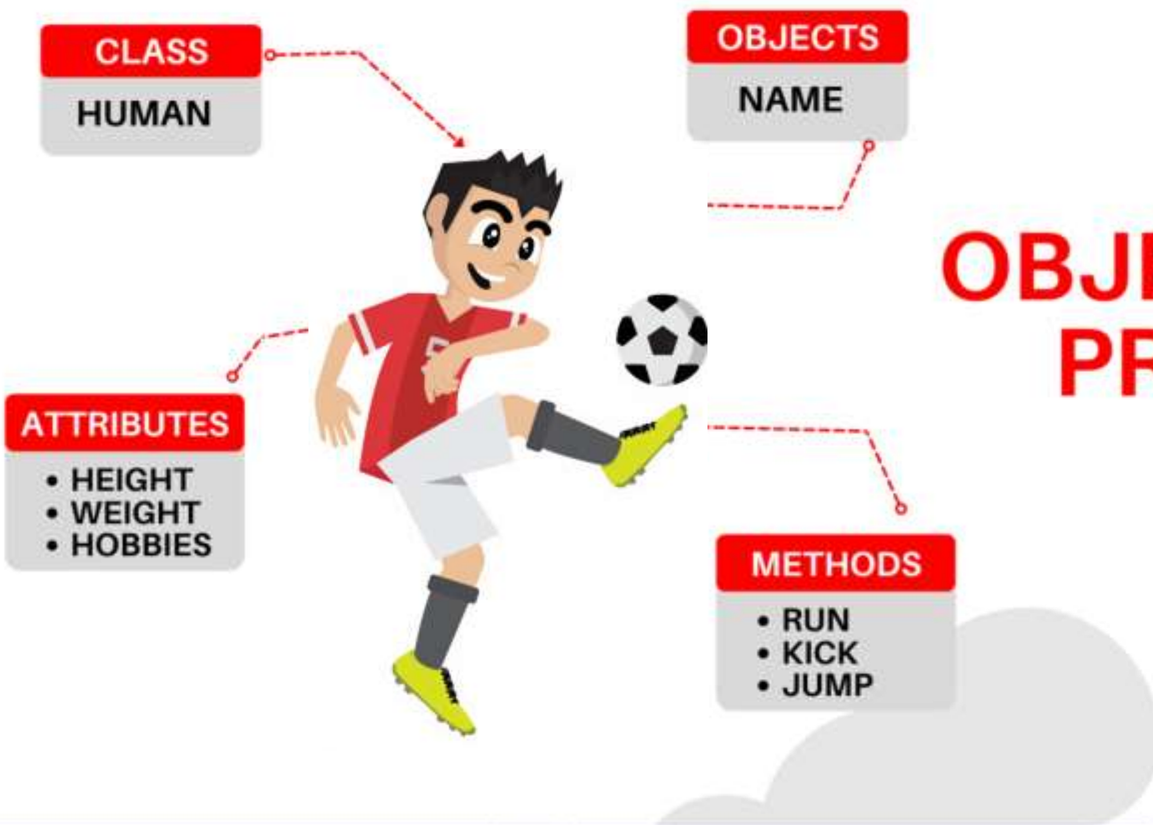
National University of Computer and Emerging Sciences



OBJECT-ORIENTED PRORAMMING

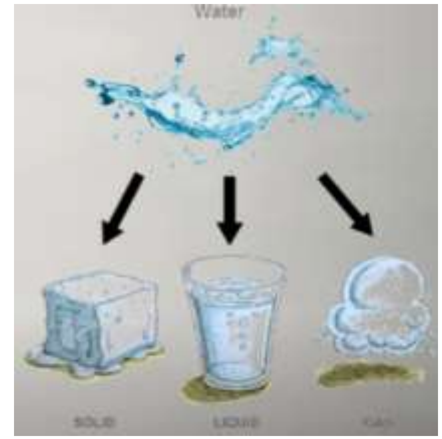
Summer 2023

Pir Sami Ullah Shah

Lecture # 10 Polymorphism

Polymorphism

- Combination of two Greek words
  - Poly (many) morphism (form)



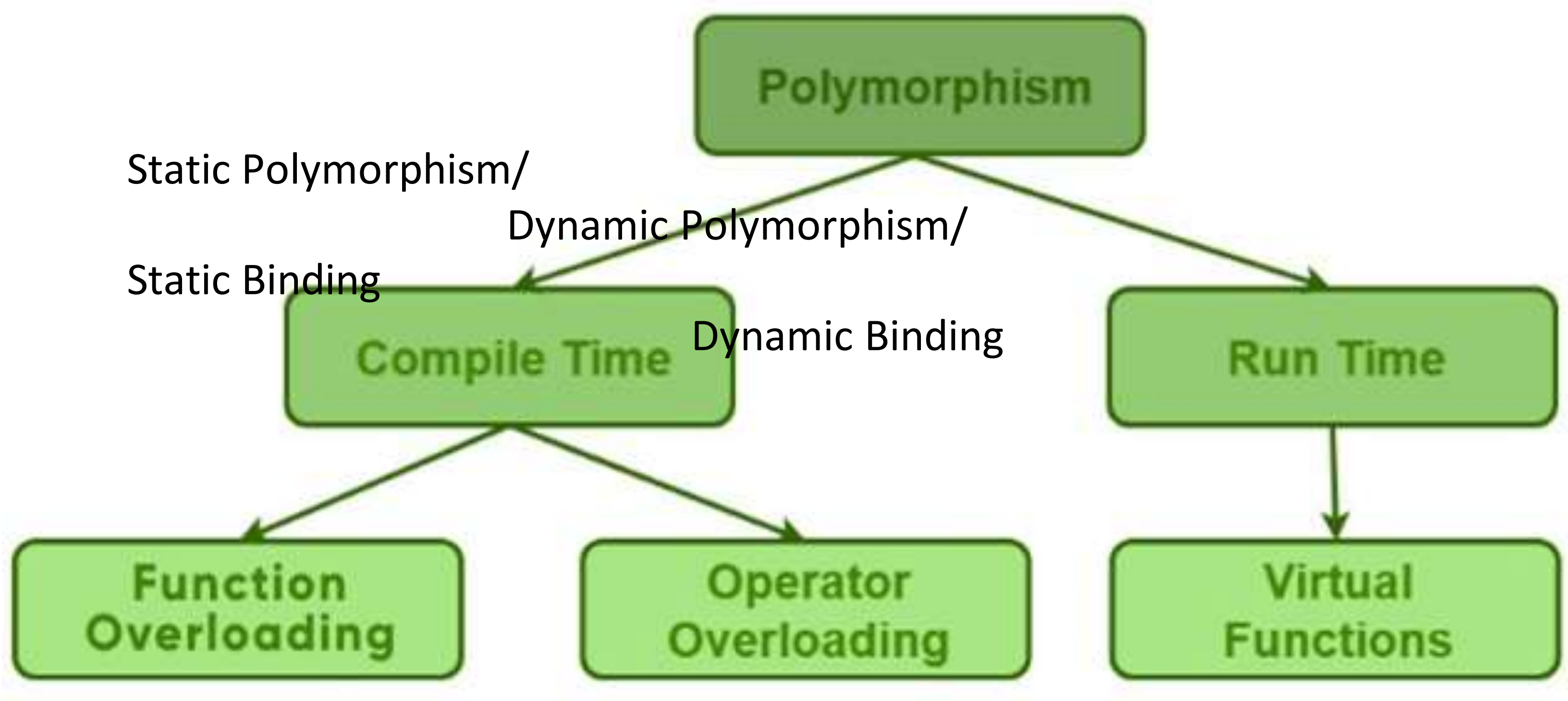
- Water -> Solid, Liquid, Gas

- For example, A Person

In shopping mall, behaves like a customer
In metro bus, behaves like a passenger
In university, behaves like a student
In home, behaves like a daughter/son

- Same person have different behavior in different situations. This is called Polymorphism.

Polymorphism



Binding Process

- Binding is the process to associate variable/ function names with memory addresses

- Binding is done for each variable and functions.

- For functions, it means that matching the call with the right function definition by the compiler.

Compile-time Binding (Static Binding)

- Compile-time binding is to associate a function's name with the entry point (start memory address) of the function at compile time (also called early binding)

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

void sayHi();
int main(){
    sayHi();      // the compiler binds any invocation of sayHi()
                  // to sayHi()'s entry point.
}
void sayHi(){
    cout << ``Hello, World!\n'';
}
```

Start address if sayHi() function

Run-time Binding (Dynamic Binding)

- Run-time binding is to associate a function's name with the entry point (start memory address) of the function at run time (also called late binding)

- C++ provides both compile-time and run-time bindings:
  - Non-Virtual functions (you have implemented so far) are binded at compile time
  - Virtual functions (in C++) are binded at run-time.

- Why virtual functions are used?
  - To implement Polymorphism

Static Polymorphism

```
//Function Overloading
class SomeClass
{
    public:
    // function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }
    // function with same name but 1 double parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }
    // function with same name and 2 int parameters
    void func(int x, int y)
    {
        cout << "value of x and y is " << x << ", " << y << endl;
    }
};
```

```
int main() {

    SomeClass obj1;
    // The first 'func' is called
    obj1.func(7);

    // The second 'func' is called
    obj1.func(9.132);

    // The third 'func' is called
    obj1.func(85,64);
    return 0;
}
```

Dynamic Polymorphism

- There is an inheritance hierarchy

- There is a pointer/reference of base class type that can point/refer to derived class objects

Pointers to Derived Classes

- C++ allows base class pointers or references to point/refer to both base class objects and also all derived class objects.

- Let's assume:

```
class Base { … };
class Derived : public Base { … };
```

- Then, we can write:

```
Base *p1;
Derived d_obj;      p1 = &d_obj;
Base *p2 = new Derived;
```

Pointers to Derived Classes (contd.)

- While it is allowed for a base class pointer to point to a derived object, the reverse is not true.

```
base b1;
derived *pd = &b1; // compiler error
```

Pointers to Derived Classes (contd.)

- Access to members of a class object is determined by the type of
  - An object name (i.e., variable, etc.)
  - A reference to an object
  - A pointer to an object

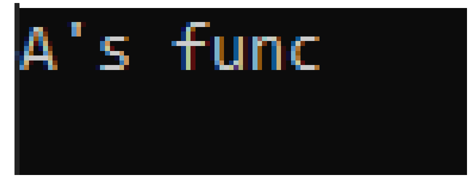
Pointers to Derived Classes (contd.)

- Using a base class pointer (pointing to a derived class object) can access only those members of the derived object that were inherited from the base.

- This is because the base pointer has knowledge only of the base class.

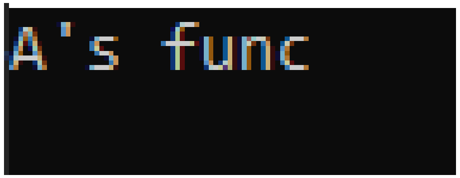
- It knows nothing about the members added by the derived class.

Pointer of Base Class




```
class A {
public:
        void func() {
                cout << "A's func" << endl;
        }
};
class B:public A {
public:
        void func() {
                cout << "B's func" << endl;
        }
};
void main() {
        B b;
        //pointer of class type A points to
        //object of child class B
        A* a = &b;
        a->func(); //calls A's func
}
```

Reference of Base Class




```
class A {
public:
        void func() {
                cout << "A's func" << endl;
        }
};
class B:public A {
public:
        void func() {
                cout << "B's func" << endl;
        }
};
void main() {
        B b;
        //reference of class type A refers to
        //object of child class B
        A& a = b;
        a.func(); //calls A's func
}
```

Pointer of Base Class

Parent class pointer/reference **has NO KNOWLEDGE** of child class functions

```
class A {
public:
        void func() {
                cout << "A's func" << endl
        }
};
class B:public A {
public:
        void func()  {
                cout << "B's func" << endl;
        }
        void foo() {}
};

void main() {
        //pointer of class type A points to
        //object of child class B
        A* a = new B;
        a->foo(); //ERROR
}
```

Summary – Based and Derived Class Pointers

- Base-class pointer pointing to base-class object
  - Straightforward

- Derived-class pointer pointing to derived-class object
  - Straightforward

- Base-class pointer pointing to derived-class object
  - Safe
  - Can access non-virtual methods of only base-class
  - Can access virtual methods of derived class

- Derived-class pointer pointing to base-class object
  - Compilation error

Dynamic Polymorphism

- There is an inheritance hierarchy
- There is a pointer/reference of base class type that can point/refer to derived class objects
- There is a pointer of base class type that is used to invoke virtual functions of derived class.
- The first class that defines a virtual function is the base class of the hierarchy that uses dynamic binding for that function name and signature.
- Each of the derived classes in the hierarchy must have a virtual function with same name and signature. Not an error but needed for dynamic binding

Virtual Functions

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call

- They are mainly used to achieve Runtime polymorphism

- Functions are declared with a virtual keyword in base class

- The resolving of function call is done at runtime

Virtual Functions

- The virtual-ness of an operation is always inherited
- If a function is virtual in the base class, it must be virtual in the derived class
- Even if the keyword “virtual” not specified (But always use the keyword in children classes for clarity.)
  - If no overridden function is provided, the virtual function of base class is used

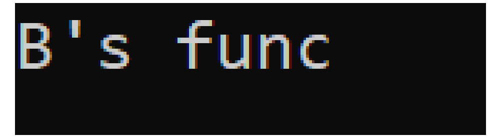
Virtual function

- Declaring a function virtual will ensure late-binding

- To declare a function virtual, we use the Keyword virtual:

```
class Shape
{
  public:
        virtual void sayHi ()
        {
              cout <<"Just hi! \n";
        }
};
```

Virtual function

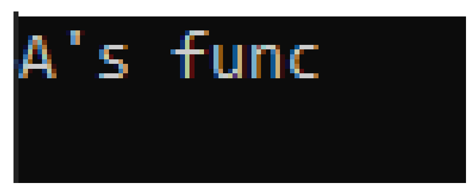



```
class A {
public:
	virtual void func() {
		cout << "A's func" << endl;
	}
};
class B:public A {
public:
	void func() { //automatically virtual
		cout << "B's func" << endl;
	}
};
void main() {
B b;
//pointer of class type A points to
//object of child class B
A* a = &b;
a->func(); //calls B's func
}
```

Virtual function

```cpp
class A {
public:
        virtual void func() {
                cout << "A's func" << end
        }
};
class B:public A {
public:
        void func(int a) { //automatically virtual
                //BUT, parameters don't match base class func()
                cout << "B's func" << endl;
        }
};
void main() {
        B b;
        //pointer of class type A points to
        //object of child class B
        A* a = &b;
        a->func(); //calls A's func
}
```

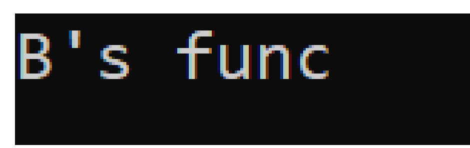
Overridden function parameters in derived class **must be same as base class**, otherwise base class func will be called



Virtual function

Override keyword

```
class A {
public:
        virtual void func() {
                cout << "A's func" << endl;
        }
};
class B:public A {
public:
        void func() override { //automatically virtual
                //use override keyword to ensure parameters match
                cout << "B's func" << endl;
        }
};
void main() {
B b;
//pointer of class type A points to
//object of child class B
A* a = &b;
a->func(); //calls B's func
}
```



Virtual function with Multilevel Inheritance

```
class A {
public:
	virtual void func() {
		cout << "A's func" << endl;
	}
};
class B:public A {
public:
	void func() override { //automatically virtual
		//use override keyword to ensure parameters match
		cout << "B's func" << endl;
	}
};
class C :public B {
public:
	void func() override { //automatically virtual
		//use override keyword to ensure parameters match
		cout << "C's func" << endl;
	}
};
void main() {
	//pointer of class type A points to
	//object of grandchild class C
	A* a = new C;
	a->func(); //calls C's func
}
```

Virtual function with Multilevel Inheritance

```
class A {
public:
        virtual void func() {
                cout << "A's func" << endl;
        }
};
class B:public A {
public:
        void func() override { //automatically virtual
                //use override keyword to ensure parameters match
                cout << "B's func" << endl;
        }
};
class C :public B {};
void main() {
        //pointer of class type A points to
        //object of grandchild class C
        A* a = new C;
        a->func(); //calls B's func
}
```

Class C does not override func(), if parent of class C has func(), that one is executed

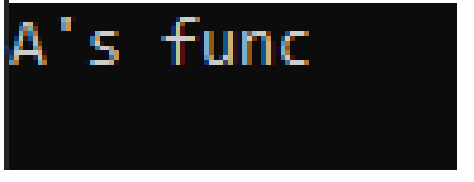
```
B's func
```

Virtual function with Multilevel Inheritance

```
class A {
public:
        virtual void func() {
                cout << "A's func" << en
        }
};
class B:public A {};
class C :public B {};

void main() {
        //pointer of class type A points to
        //object of grandchild class C
        A* a = new C;
        a->func(); //calls A's func
}
```

Class C does not override func(), class B also does not override func(),
A's func is executed



Virtual Functions

- If the member function definition is out-of-line, the keyword virtual must not be specified again.

```
class Shape{
public:
  virtual void sayHi ();
};
virtual void Shape::sayHi (){  // error
  cout << ``Just hi! \n'';
}
```

- Virtual functions can not be stand-alone or static functions

- A destructor can be virtual but a constructor cannot

Virtual Functions based Shapes

```
class Shape{
public:
   virtual void sayHi() { cout <<''Just hi! \n'';}
};
class Triangle : public Shape{
public:
   virtual void sayHi() { cout <<''Hi from a triangle! \n'';}
};
class Rectangle : public Shape{
public:
   virtual void sayHi() { cout <<''Hi from a rectangle! \n; }
};

int main(){
   Shape *p;
   int which;
   cout << ''1 -- shape, 2 -- triangle, 3 -- rectangle\n '';
   cin >> which;
   switch ( which ) {
   case 1:  p = new Shape; break;
   case 2:  p = new Triangle; break;
   case 3:  p = new Rectangle; break;
   }
   p -> sayHi();    // dynamic binding of sayHi()
   delete p;
}
```

Virtual Functions

How to declare a member function virtual:

```
class Animal{
  public:
      virtual void id(){cout << "animal";}
};

class Cat : public Animal{
  public:
      virtual void id(){cout << "cat";}
};

class Dog : public Animal{
  public:
      virtual void id(){cout << "dog";}
};
```

Virtual Functions

- If the member functions ***id( )*** are declared ***virtual***, then the code:

```
Animal *pA[] = {new Animal, new Dog, new Cat};

for(int i=0; i<3; i++)
    pA[i]->id();
```

will print ***animal, dog, cat***

With Multiple Inheritance

```
class A {
public:
         void print() { //not virtual
        cout << "Print class A" << endl;
    }
         ~A() {
             cout << "A's destructor" << endl;
         }
};
class B  {
public:
      virtual void print()   {
        cout << "Print class B" << endl;
    }
      ~B() {
          cout << "B's destructor" << endl;
      }
};
class C :public B,public A {
public:
     void print() {
        cout << "Print class C" << endl;
    }
    ~C() {
        cout << "C's destructor" << endl;
    }
};
```

```
int main() {
    B *b=new C;
    A* a=new C;
    b->print();
        a-
>print();
    return 0;
}
```

```
Output:
Print class C
Print class A
```

Benefits of Polymorphism

Better Design!

Flexibility

- You can always change the subclass object assigned to the superclass reference variable, without breaking other code
- The modification will only affect the new object, not those using it

Need to Write Less code

- Reference variable of superclass type can be assigned object of any subclass

Easy to Extend

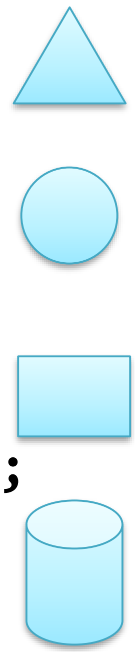
- Write code that doesn't have to change when you introduce new subclass types into the program.

Pointers to Derived Classes

- We can create an array of base class pointers, and these pointers can hold objects of different derived classes

```
Shape *p[4];
p[0] = new Triangle (3, 4, 5, 19 );
p[1] = new Circle (3, 4, 5 );
p[2] = new Rectangle ( 3, 4, 10 , 20 );
p[3] = new Cylinder ( 3, 4, 5, 10 );

for ( int loop = 0; loop < 4; loop ++ )
{    p[loop]->draw ();
     cout << “The area is “ << p[loop]->GetArea ( );
}
```



Dynamic Polymorphism Example (using Base Class’s Pointers and References)

```
class Shape{
public:
   virtual void sayHi() { cout <<``Just hi! \n'';}
};
class Triangle : public Shape{
public:
   // overrides Shape::sayHi(), automatically virtual
   void sayHi() { cout <<``Hi from a triangle! \n'';}
};

void print(Shape obj, Shape *ptr, Shape &ref){
   ptr -> sayHi();    // bound at run time
   ref.sayHi();       // bound at run time
   obj.sayHi();       // bound at compile time
}

int main(){
  Triangle mytri;
  print( mytri, &mytri, mytri );
}
```

Virtual Destructors

- Constructors cannot be virtual, but destructors can be virtual when a constructor of a class is executed there is no virtual table in the memory, means no virtual pointer defined yet.

- ***Ensures the derived class destructor is called when a base class pointer is used,*** while deleting a dynamically created derived class object.

  virtual ~Shape(){….}

  – Reason: to invoke the correct destructor, no matter how object is accessed

Virtual Destructors (contd.)

```
class base {
public:
    ~base() {
        cout <<  "destructing
   base\n";
    }
};

class derived : public base {
public:
    ~derived() {
        cout << "destructing
   derived\n";
    }
};
```

```
int main()
{
    base *p = new derived;
    delete p;
   return 0;
}
```

Output:

destructing base

Using non-virtual destructor

Virtual Destructors (contd.)

```
class base {

public:
   virtual ~base() {
      cout <<  “destructing
   base\n”;
   }
};

class derived : public base {
public:
   ~derived() {
      cout << “destructing
   derived\n”;
   }
};
```

```
int main()
{
    base *p = new derived;
    delete p;

   return 0;
}
```

Output:

```
    destructing derived
    destructing base
```

Using virtual destructor

```cpp
class A {
public:
        void print(int b) {
        cout << "Print class A" << endl;
    }
        ~A() {
            cout << "A's destructor" << endl;
        }
};
class B :public  A {
public:
     void print(int a=0)   {
       cout << "Print class B" << endl;
    }
     ~B() {
        cout << "B's destructor" << endl;
     }
};
class C :public B {
public:
    ~C() {
        cout << "C's destructor" << endl;
    }
};
int main() {
    B *b=new B;
    A* a=new A;
    delete a;
    delete b;
    B bb;
    C c;
    A aa;
    return 0;
}
```

```
A's destructor
B's destructor
A's destructor
A's destructor
C's destructor
B's destructor
A's destructor
B's destructor
A's destructor
```

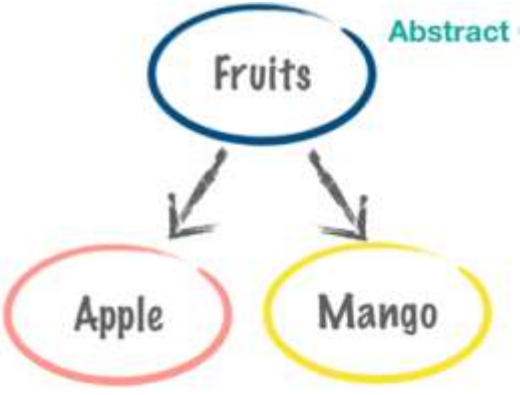
- For dynamic objects, destructors are called with delete only and in the order of delete statements.

- For simple objects (in the same scope) destructors are called in opposite order. i.e. the one declared last is destroyed first.

- Without delete, destructor is not called for dynamic objects

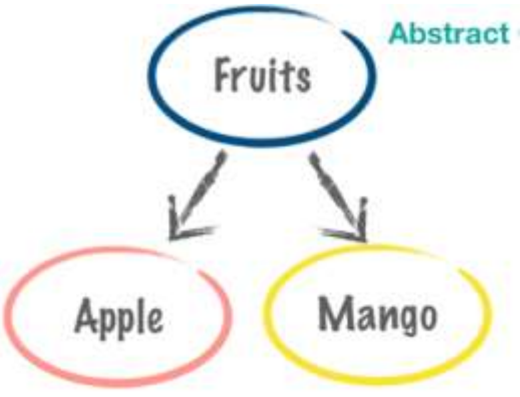
Abstract Classes

- Classes that ***cannot be instantiated (a class with no objects), because:***
  - It is ***Incomplete***—derived classes must define the “missing pieces”
  - Too generic to define real objects
- Normally used as base classes and called abstract base classes



Concrete Classes

- Classes that can be instantiated (have objects)
- Must provide implementation for every member function they define



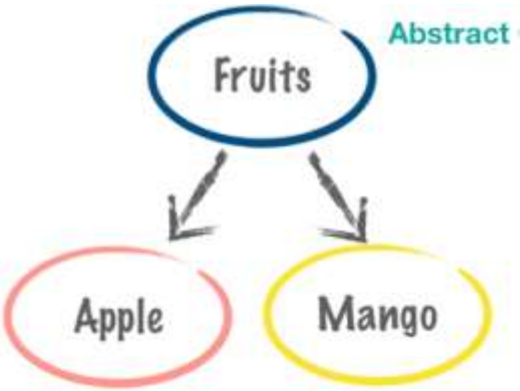
Pure virtual Functions

- A class is made abstract by declaring one or more of its virtual functions to be “***pure***”
  - I.e., by placing "**= 0**" in its declaration

- Example:

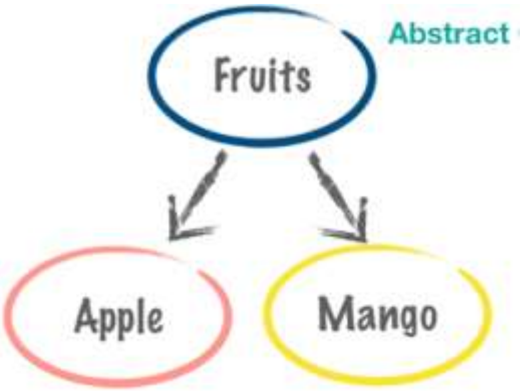
```
virtual void draw() = 0;
```

  - "= 0" is known as a pure specifier.
  - Tells compiler that there is no implementation.



Pure virtual Functions (cont.)

- Every concrete derived class must override all base-class pure virtual functions
  - with concrete implementations
- If even one pure virtual function is not overridden
  - the ***derived-class will also be abstract***
  - Compiler will refuse to create any objects of the class
  - Cannot call a constructor



Pure virtual Functions (cont.)

```
class A { //abstract class
public:
        //pure virtual function
        virtual void func() = 0;
        void foo() {
                cout << "A's foo" << endl;
        }
};
class B:public A {
public:
        void func()  { //automatically virtual
                cout << "B's func" << endl;
        }
};
void main() {
        A objA; //ERROR, cannot create object of abstract class
        A* a = new B;  //dynamic polymorphism
        a->func(); //calls B's func
}
```

A class with **even one pure virtual function** is an **abstract class**

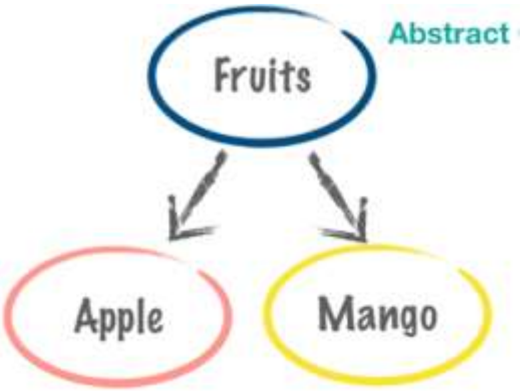
Pure virtual Functions (cont.)

```
class A { //abstract class
public:
        //pure virtual function
        virtual void func() = 0;
        void foo() {
                cout << "A's foo" << endl;
        }
};
class B:public A {
public:
        //does not override func() also an abstract class now
};
void main() {
        A* a = new B;  //ERROR, B is abstract
}
```

Purpose

- When it does not make sense for base class to have an implementation of a function

- Software design requires *all concrete derived classes to implement their own function*



Why Do we Want to do This?

- To define a common public interface for the various classes in a class hierarchy
  - Achieve *dynamic polymorphism*

- The heart of object-oriented programming

- Simplifies a lot of big software systems
  - Enables code re-use in a major way
  - Readable, maintainable, adaptable code