FROM C TO C++

OBJECT PROGRAMMING

-PART II-

HISTORY

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as **attributes**; and code, in the form of procedures, often known as **methods**.
- There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that **objects are instances of classes**, which typically also determine their *type*.
- Significant object-oriented languages include Java, C++, C#, Python, PHP, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, Common Lisp, and Smalltalk.

PROTOTYPE-BASED

- A different OOP paradigm: not based on classes
  - ✓Javascript, Self
- Behaviour reuse (known as inheritance) is performed via a process of reusing existing objects via delegation that serve as prototypes
  - ✓*Objects inherit from objects*
- Let's use the idea of "fruit" as one example. A "fruit" object would represent the properties and functionality of fruit in general. A "banana" object would be cloned from the "fruit" object, and would also be extended to include general properties specific to bananas. Each individual "banana" object would be cloned from the generic "banana" object. Compare to the class-based paradigm, where a "fruit" *class* (not object) would be extended by a "banana" *class*.

DELEGATION

- In object-oriented programming, **delegation** refers to evaluating a member (property or method) of one object (the receiver) in the context of another, original object (the sender).
- Implicit delegation is the fundamental method for behavior reuse in prototype-based programming, corresponding to inheritance in class-based programming.

EXAMPLE

```
// Example of true prototypal inheritance style in JavaScript

var foo = {name: "foo", one: 1, two: 2}; // object notation {}.
var bar = {two: "two", three: 3}; // Another object.


// Object.setPrototypeOf() is a method introduced in ECMAScript 2015.
// For the sake of simplicity, let us pretend that the following line works regardless of the
// engine used:

// foo is now the prototype of bar.
// If we try to access foo's properties from bar from now on, we'll succeed.
Object.setPrototypeOf(bar, foo);

bar.one // Resolves to 1.

// The child object's properties are also accessible.
bar.three // Resolves to 3. /
// Own properties shadow prototype properties
bar.two; // Resolves to "two"
bar.name; // unaffected, resolves to "foo"
foo.name; // Resolves to "foo"
```

HISTORY

- Terminology invoking "objects" and "oriented" in the modern sense of object-oriented programming made its first appearance at MIT in the late 1950s and early 1960s
- The formal programming concept of objects was introduced in the mid-1960s with Simula 67, a major revision of Simula I, a programming language designed for discrete event simulation, created by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center in Oslo
- The Smalltalk language, which was developed at Xerox PARC (by Alan Kay and others) in the 1970s, introduced the term object-oriented programming to represent the pervasive use of objects and messages as the basis for computation.

HISTORY

- Object-oriented programming developed as the dominant programming methodology in the early and mid 1990s when programming languages supporting the techniques became widely available
- Its dominance was further enhanced by the rising popularity of graphical user interfaces, which rely heavily upon object-oriented programming techniques
- Object-oriented features have been added to many previously existing languages,
  including Ada, BASIC, Fortran, Pascal, and COBOL
  - ✓Adding these features to languages that were not initially designed for them often led to problems with compatibility and maintainability of code

ENCAPSULATION

- Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse
  - ✓Data encapsulation led to the important OOP concept of **data hiding**
- Encapsulation prevents external code from being concerned with the internal workings of an object
- This facilitates code refactoring, for example allowing the author of the class to change how objects of that class represent their data internally without changing any external code (as long as "public" method calls work the same way).
- Encapsulation is a technique that encourages **decoupling**
  - ✓to put all the code that is concerned with a certain set of data in the same class

EXAMPLE OF ENCAPSULATION

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

class Adder{
public:
// constructor
Adder(int i = 0) { total = i; }

// interface to outside world
void addNum(int number) { total += number; }
int getTotal() { return total; };

private:
// hidden data from outside world
int total; };
```

```
int main( ) {
Adder a;
a.addNum(10);
a.addNum(20);
a.addNum(30);
 cout << "Total " << a.getTotal() <<endl;
return 0;
}
```

COMPOSITION

- Objects can contain other objects in their instance variables; this is known as object composition
- Object composition is used to represent "has-a" relationships

```
class Band
{
   Guitarist gtr;
   Drummer drm;
   Vocalist voc;
   ...
};
```

```
class Guitarist
{...};
```

```
class Drummer
{...};
```

```
class Drummer
{...};
```

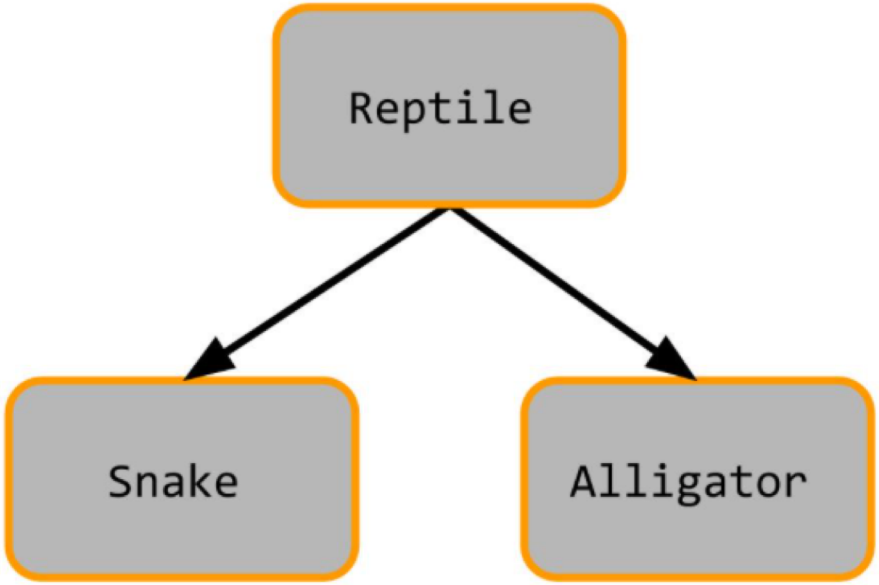
EXAMPLE

```
#include <iostream>
#include <string>

using namespace std;

class Birthday{
public:
Birthday(int cmonth, int cday, int cyear){
cmonth = month;
cday = day;
cyear = year;
}

void printDate(){
cout<<month <<"/" <<day <<"/" <<year <<endl;
}

private:
int month; int day; int year;
};
```

```
class People{
public:
People(string cname, Birthday
cdateOfBirth) : name(cname),
dateOfBirth(cdateOfBirth) { }

void printInfo(){
cout<<name <<" was born on: ";
dateOfBirth.printDate(); }

private:
string name;
Birthday dateOfBirth;
};
```

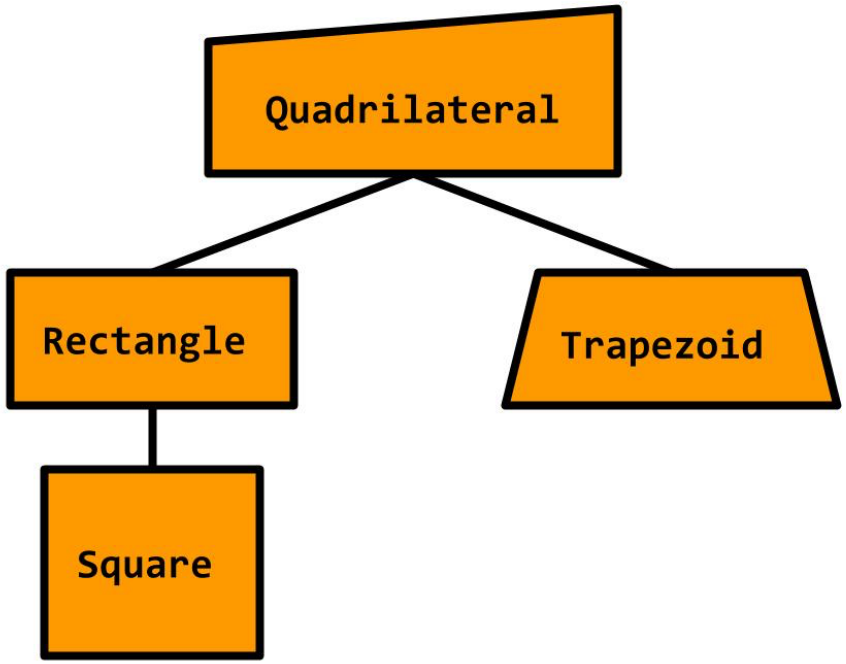
INHERITANCE

- Languages that support classes almost always support inheritance. This allows classes to be arranged in a hierarchy that represents "is-a-type-of" relationships.
- This technique allows easy re-use of the same procedures and data definitions, in addition to potentially mirroring real-world relationships in an intuitive way.



INHERITANCE

- A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes
- This existing class is called the base class, and the new class is referred to as the derived class



EXAMPLE

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

// Base class
class Shape {

public:
void setWidth(int w) { width = w; }
void setHeight(int h) { height = h; }

protected: int width; int height;
};

// Derived class
class Rectangle: public Shape {
public: int getArea() { return (width * height); }
};
```

```
int main(void) {
Rectangle Rect;
Rect.setWidth(5);
Rect.setHeight(7);

// Print the area of the object.
cout << "Total area: " << Rect.getArea() <<
endl; r
eturn 0;
}
```

Total area: 35

ACCESS CONTROL AND INHERITANCE

- A derived class can access all the non-private members of its base class.

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

ACCESS CONTROL AND INHERITANCE

- A derived class inherits all base class methods with the following exceptions:
  - ✓Constructors, destructors and copy constructors of the base class
  - ✓Overloaded operators of the base class
  - ✓The friend functions of the base class

TYPE OF INHERITANCE

- When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance
  - ✓**Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class
  - ✓**Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class
  - ✓**Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class

EXAMPLE

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

// Base class Shape
class Shape {
public:
void setWidth(int w) { width = w; }
void setHeight(int h) { height = h; }
protected: int width; int height;
};

// Base class PaintCost
class PaintCost {
public:
int getCost(int area) { return area * 70; }
};
```

```
// Derived class
class Rectangle: public Shape, public
PaintCost {

public: int getArea() { return (width * height); }
};

int main(void) {
Rectangle Rect;
int area; Rect.setWidth(5);
Rect.setHeight(7);
area = Rect.getArea();

// Print the area of the object.
cout << "Total area: " << Rect.getArea() <<
endl;

// Print the total cost of painting cout <<
"Total paint cost: $" << Rect.getCost(area)
<< endl;
return 0;
}
```

ONE MORE EXAMPLE

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

 class B : public A {
// x is public
// y is protected
// z is not accessible from B };

class C : protected A {
// x is protected
// y is protected
// z is not accessible from C };

class D : private A // 'private' is default for classes {
// x is private
 // y is private
// z is not accessible from D
};
```

COPY CONSTRUCTOR

- The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:
  - ✓Initialize one object from another of the same type
  - ✓Copy an object to pass it as an argument to a function
  - ✓Copy an object to return it from a function
- If a copy constructor is not defined in a class, the compiler itself defines one
- If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor
- *class_name* (const *class_name* &)

EXAMPLE

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

xclass Line {
public:
int getLength( void );
Line( int len );  // simple constructor
Line( const Line &obj); // copy constructor
~Line();

private:
int *ptr;
};

Line::Line(int len) {
cout << "Normal constructor allocating ptr" << endl;
ptr = new int;
*ptr = len;
}

Line::Line(const Line &obj) {
cout << "Copy constructor allocating ptr." << endl;
ptr = new int;
*ptr = *obj.ptr; }

Line::~Line(void) {
cout << "Freeing memory!"
<< endl;
delete ptr;
}
```

OVERLOADING

- In programming, operator overloading, sometimes termed operator ad-hoc polymorphism, is a specific case of polymorphism, where different operators have different implementations depending on their arguments
- Operator overloading has often been criticized[2] because it allows programmers to reassign the semantics of operators depending on the types of their operands. For example, the use of the << in C++'s
  - ✓a << 1
- Because operator overloading allows the original programmer to change the usual semantics of an operator and to catch any subsequent programmers by surprise, it is considered good practice to use operator overloading with care
  - ✓The creators of Java decided not to use this feature

EXAMPLE (FUNCTION OVERLOADING)

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

class printData {
public:
void print(int i) { cout << "Printing int: " << i
<< endl; }
void print(double f) { cout << "Printing float:
" << f << endl; }
void print(char* c) { cout << "Printing
character: " << c << endl; }
};

int main(void) { printData pd;
pd.print(5);
pd.print(500.263);
pd.print("Hello C++");
return 0;
}
```

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

OPERATOR OVERLOADING

- A programmer can provide his or her own operator to a class by overloading the built-in operator to perform some specific computation when the operator is used on objects of that class.
- Less code, more elegant
- On the other hand, operator overloading, like any advanced C++ feature, makes the language more complicated.

EXAMPLE (OPERATOR OVERLOADING)

```
Complex a(1.2,1.3);    //this class is used to represent complex numbers
Complex b(2.1,3);
Complex c = a+b;       //for this to work the addition operator must be overloaded

    // The addition without having overloaded operator + could look like this:
    Complex c = a.Add(b);
```

```
class Complex {
public:
Complex(double re,double im) :real(re),imag(im) {};
Complex operator+(const Complex& other);
Complex operator=(const Complex& other);

private:
double real; double imag;
};

Complex Complex::operator+(const Complex& other) {
double result_real = real + other.real;
double result_imaginary = imag + other.imag;
return Complex( result_real, result_imaginary );
}
```

SYNTACTIC SUGAR

- Operator overloading is syntactic sugar
- In computer science, syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express
- It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer
  - ✓ In the C language, the a[i] notation is syntactic sugar for *(a + i)
  - ✓ Likewise, the a->x notation is syntactic sugar for accessing members using the dereference operator (*a).x
  - ✓ Compound assignment operators: for example, a += b is equivalent to a = a + b in C

FRIEND

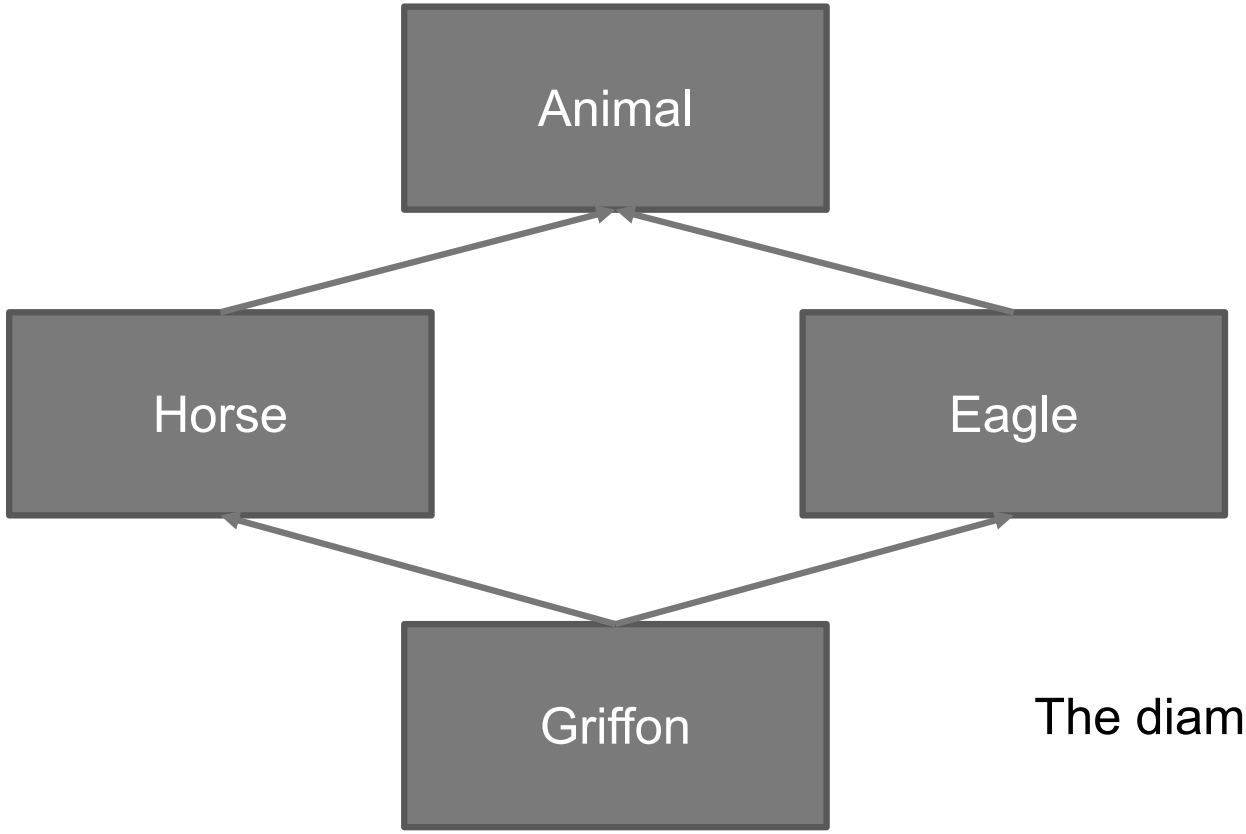
- In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to "friends"
- Friends are functions or classes declared with the friend keyword
- A non-member function can access the private and protected members of a class if it is declared a friend of that class
- That is done by including a declaration of this external function within the class, and preceding it with the keyword friend
- Typical use cases of friend functions are operations that are conducted between two different classes accessing private or protected members of both.

EXAMPLE

```
#include <iostream>
using namespace std;
class Rectangle {
int width, height;

public: Rectangle() {}
Rectangle (int x, int y) : width(x), height(y) {}
int area() {return width * height;}
friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param) {
Rectangle res;
res.width = param.width*2;
res.height = param.height*2;
return res;
}

int main () {
Rectangle foo;
Rectangle bar (2,3);
foo = duplicate (bar);
cout << foo.area() << '\n';
return 0;
}
```

MULTIPLE INHERITANCE

- A C++ class can inherit members from more than one class and here is the extended syntax:
  - ✓class derived-class: access baseA, access baseB....



The diamond problem!!

PITFALLS

- The most difficult to avoid complication that arises when using multiple inheritance is that sometimes the programmers interested in using this technique to extend the existing code are forced to learn some of the implementation's details
- The second trivial problem that might appear when using this technique is the creation of ambiguities:

```
class A {void f(); };
class B {void f(); };
class C : public A ,public B { void f(); };

C* pc = new C;
pc->f();
pc->A::f();   //this calls f() from class A
pc->B::f();   //this calls f() from class B
```

PITFALLS

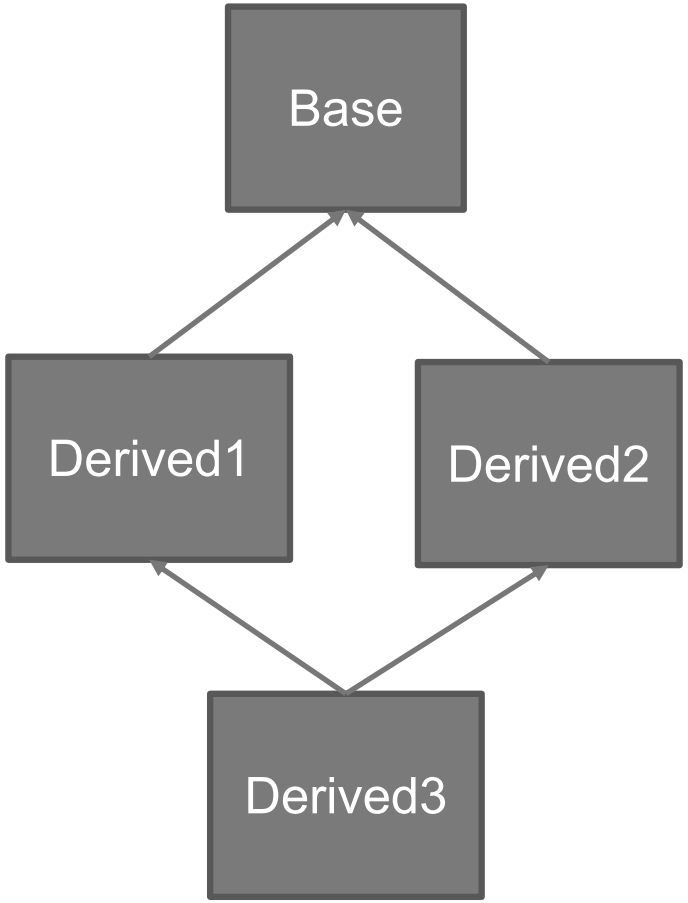
- If the C class didn't have the method f() the problem couldn't have been solved with explicit qualification. Instead we would have used implicit conversion:

```
A* pa = pc; pc->f();
```

- or we would have to make a cast in order to call the method from the parent class A.

DEADLY DIAMOND OF DEATH

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

class Base {
protected:
int iData;
public:
Base() { iData = 10;}
};

class Derived1 : public Base {};
class Derived2 : public Base {};

class Derived3 : public Derived1, public Derived2 {
public:
int GetData() { return iData;}
};

int main (){
Derived3 obj;
cout << obj.GetData() << endl;
}
```



DIAMOND PROBLEM

- If the two classes derived from the same base class, and that base class has one or more members, then those members will be duplicated in the joining class.

```
MacBook-Francesco:c++ francescosantini$ g++ -o diamond
diamond.cpp
diamond.cpp:47:20: error: non-static member 'iData' found
in multiple base-class
      subobjects of type 'Base':
    class Derived3 -> class Derived1 -> class Base
    class Derived3 -> class Derived2 -> class Base
          return iData;
                 ^
diamond.cpp:13:13: note: member found by ambiguous name
lookup
       int iData;
           ^
1 error generated.
```

DIAMOND PROBLEM

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

class Base {
public:
Base() {}
void foo() {cout << "1";}
};

class Derived1 : public Base {
public:void foo() {cout << "2";}
};

class Derived2 : public Base {
public:void foo() {cout << "3";}
};

class Derived3 : public Derived1, public Derived2 {};

int main (){
Derived3 obj;
cout << obj.foo() << endl;}
```

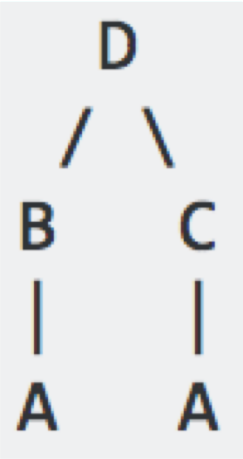
DIAMOND PROBLEM

```
MacBook-Francesco:c++ francescosantini$ g++ -o diamond
diamond.cpp
diamond.cpp:49:17: error: member 'foo' found in multiple
base classes of
      different types
    cout << obj.foo() << endl;
                ^
diamond.cpp:24:6: note: member found by ambiguous name
lookup
void foo() {cout << "2";}
     ^
diamond.cpp:31:6: note: member found by ambiguous name
lookup
void foo() {cout << "2";}
     ^
1 error generated.
```

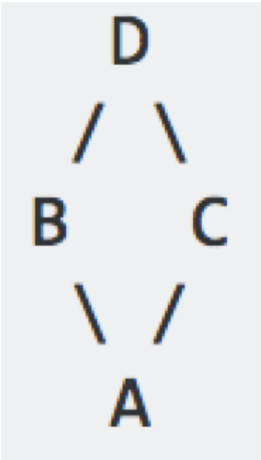
A SOLUTION: VIRTUAL INHERITANCE

- Using the keyword virtual in the derived classes (Derived1, Derived2) resolves this ambiguity by allowing them to share a single instance of base class.

```
class D { public: void Foo() {} };
class B : public D {};
class C : public D {};
class A : public B, public C {};
```




```
class D { public: void Foo() {} };
class B : public virtual D {};
class C : public virtual D {};
class A : public B, public C {};
```



ABSTRACT CLASSES (INTERFACES)

- The purpose of an **abstract class** is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.
- A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows:

```
class Box {
public:
  // pure virtual function
  virtual double getVolume() = 0;
private:
  double length; // Length of a box
  double breadth; // Breadth of a box
  double height; // Height of a box };
```

VIRTUAL METHODS

- Classes that can be used to instantiate objects are called **concrete classes**
- A pure virtual function or pure virtual method is a virtual function that is required to be implemented by a derived class if the derived class is not abstract
- A derived of an abstract class can only be instantiated directly if all inherited pure virtual methods have been implemented by that class or a parent class
- Virtual methods can also be used where the method declarations are being used to define an interface - similar to what the **interface** keyword in Java explicitly specifies.

EXAMPLE OF OVERRIDE

- In a member function declaration or definition, override ensures that the function is virtual and is overriding a virtual function from the base class.

```
class Animal {
public:
void /*non-virtual*/ move(void) {
std::cout << "This animal moves in some way" << std::endl; }

virtual void eat(void) = 0;
  };

class Llama : public Animal {
public:
// The non virtual function move() is inherited but not overridden
void eat(void) override {
std::cout << "Llamas eat grass!" << std::endl; }
};
```

A PROBLEM THEY SOLVE

- In object-oriented programming, when a derived class inherits from a base class, an object of the derived class may be referred to via a pointer or reference of the base class type instead of the derived class type. If there are base class methods overridden by the derived class, the method actually called by such a reference or pointer can be bound either 'early' (by the compiler), according to the declared type of the pointer or reference, or 'late' (i.e., by the runtime system of the language), according to the actual type of the object referred to
- Virtual functions are resolved 'late'. If the function in question is 'virtual' in the base class, the most-derived class's implementation of the function is called according to the actual type of the object referred to, regardless of the declared type of the pointer or reference. If it is not 'virtual', the method is resolved 'early' and the function called is selected according to the declared type of the pointer or reference

FINAL

- 1st use: when used in a virtual function declaration or definition, final ensures that the function is virtual and specifies that it may not be overridden by derived classes
- 2nd use: If a class or struct is marked as final then it becomes non inheritable and it cannot be used as base class

```
class Base final
{
};

class Derived : public Base
{
};

int main()
{
    Derived d;
    return 0;
}
```

error: cannot derive from ‘final’ base ‘Base’ in derived type ‘Derived’ class Derived : public Base

1ST USE

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

class Base {
public:
    virtual void myfun() final {
        cout << "myfun() in Base"; }
};

class Derived : public Base
{
    void myfun() {
        cout << "myfun() in Derived\n"; }
};

int main()
{
    Derived d;
    Base &b = d;
    b.myfun();
    return 0;
}
```

```
struct Base { virtual void foo(); };
struct A : Base { void foo() final; // A::foo is
overridden and it is the final override
void bar() final; // Error: non-virtual function
cannot be overridden or be final
};
```

prog.cpp:14:10: error: virtual function ‘virtual void Derived::myfun()’ void myfun()

prog.cpp:7:18: error: overriding final function ‘virtual void Base::myfun()’ virtual void myfun() final

OPEN RECURSION

- **this**, **self**, and **Me** are keywords used in some computer programming languages to refer to the object, class, or other entity that the currently running code is part of
- Another handy feature offered by most languages with objects and classes is the ability for one method body to invoke another method of the same object via **this**
- Every object in C++ has access to its own address through an important pointer called **this** pointer
- Friend functions do not have a **this** pointer, because friends are not members of a class

EXAMPLE

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

class Box {
public:
// Constructor definition
Box(double l = 2.0, double b = 2.0, double h = 2.0) {
cout <<"Constructor called." << endl;
length = l; breadth = b; height = h; }

double Volume() {
return length * breadth * height; }

int compare(Box box) {
return this->Volume() > box.Volume();
}

private:
double length;  // Length of a box
double breadth;  // Breadth of a box
double height; // Height of a box };
```

```
Constructor called.
Constructor called. Box2 is equal to or larger than Box1
```

```
int main(void) {
Box Box1(3.3, 1.2, 1.5); // Define box1
Box Box2(8.5, 6.0, 2.0); // Declare box2
if(Box1.compare(Box2)) {
    cout << "Box2 is smaller than Box1" <<endl;
}
else {
    cout << "Box2 is equal to or larger than
Box1" <<endl; }
return 0;
}
```

POLIMORPHISM

- In programming languages and type theory, polymorphism (from Greek πολύς, polys, "many, much" and μορφή, morphē, "form, shape") is the provision of a single interface to entities of different types
- **Ad hoc polymorphism**: when a function denotes different and potentially heterogeneous implementations depending on a limited range of individually specified types and combinations. Ad hoc polymorphism is supported in many languages using function overloading
- **Subtyping** (also called subtype polymorphism or inclusion polymorphism): when a name denotes instances of many different classes related by some common superclass
  - ✓ In the object-oriented programming community, this is often referred to as simply polymorphism.

EXAMPLE

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

class Polygon {
protected: int width, height;

public:
void set_values (int a, int b) {
   width=a; height=b; }
};

class Rectangle: public Polygon {
public:
int area() { return width*height; }
};

class Triangle: public Polygon {
public:
int area() { return width*height/2; }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
return 0;
}
```

```
20
10
```