

Inheritance

For : COP 3330.
Object oriented Programming (Using C++)
<http://www.compgeom.com/~piyush/teach/3330>

Piyush Kumar

OOP components

- ✓ Data Abstraction
 - Information Hiding, ADTs
- ✓ Encapsulation
- ✓ Type Extensibility
 - Operator Overloading
- ☑ Inheritance
 - Code Reuse
- ☑ Polymorphism

“Is a” Vs “Has a”

- Inheritance
 - Considered an “Is a” class relationship
 - e.g.: An HourlyEmployee “is a” Employee
 - A Convertible “is a” Automobile
- A class contains objects of another class as it’s member data
 - Considered a “Has a” class relationship
 - e.g.: One class “has a” object of another class as it’s data

Another Example

```
class Car : public Vehicle {
public:
    // ...
};
```

We state the above relationship in several ways:

- * Car is “a kind of a” Vehicle
- * Car is “derived from” Vehicle
- * Car is “a specialized” Vehicle
- * Car is the “subclass” of Vehicle
- * Vehicle is the “base class” of Car
- * Vehicle is the “superclass” of Car (this not as common in the C++ community)

UML

Virtual Functions

- Virtual means “overridable”
- Runtime system automatically invokes the proper member function.
- Costs 10% to 20% extra overhead compared to calling a non-virtual function call.

Virtual Destructor rule

- If a class has one virtual function, you want to have a virtual destructor.
- A virtual destructor causes the compiler to use dynamic binding when calling the destructor.
- Constructors: Can not be virtual. You should think of them as static member functions that create objects.

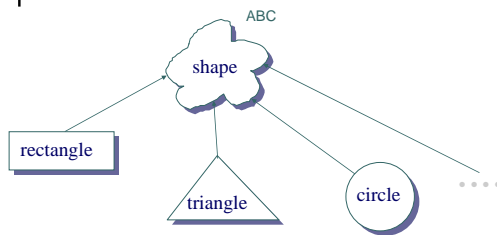
Pure virtual.

- A pure virtual member function is a member function that the base class forces derived classes to provide.
- A pure virtual function makes a class an abstract base class (ABC)
 - Can not be instantiated!
- An ABC can also have a pure virtual destructor.

Pure virtual member functions.

- Specified by writing =0 after the function parameter list.

OOP Shape Example



Public inheritance: "Is A" relationships

Abstract Base class: Shape.

```
class Shape {
public:
    Shape ( Point2d& position, Color& c ) : center_(position) , color_( c ) {};
    virtual void rotate( double radians ) = 0;
    virtual bool draw(Screen &s) = 0; // Inheritance of interface.
    virtual ~Shape(void) = 0;
    virtual void error(const string& msg); // Inheritance of implementation.
    int ObjectID() const; // Do not redefine.
    void move(Point2d& p) { _center = p; };
private:
    Point2d center_;
    Color color_;
};
```

C++ Shape example

```
class Triangle: public Shape {
public:
    Triangle( Point2d& p[3] );
    virtual void rotate ( double radians ){...}
    virtual bool draw(Screen &s) {...};
    virtual ~Triangle(void) {...};
    // Can use the default error
    // Must not define / declare ObjectID
private:
    Point2d vertices[3];
};
```

Concrete derived class

- Has no pure virtual functions.
- Simply provides the definition of all the pure virtual functions in its ABC.

Typecasts

- Can I convert a pointer of a derived class type to a base class type?
 - ?
 - Does it require a typecast?

Containers and Inheritance

- Because derived objects are “sliced down” when assigned to a base object, containers and types related by inheritance do not mix well.

```
multiset<Item_base> basket;
Item_base base;
Bulk_item bulk;
basket.insert(base);
basket.insert(bulk); // problem! (Slicing!)
```

Questions

- How can a class Y be a kind-of another class X as well as get the bits of X?
 - Is-a relationship
- How can a class Y get the bits of X without making Y a kind-of X?
 - Has a relationship

Inheritance

- Except for friendship, inheritance is the strongest relationship that can be expressed in C++, and should be only be used when it's really necessary.

Multiple Inheritance

- **Multiple inheritance** refers to a feature of [object-oriented programming languages](#) in which a [class](#) can inherit behaviors and features from more than one [superclass](#).
- Multiple inheritance can cause some confusing situations (A Diamond!)
 - Java compromises. (can inherit implementation from only one parent).
- Virtual inheritance is used to solve problems caused by MI.

Virtual and Multiple Inheritance

- Multiple and virtual Inheritance: Beyond the scope of this class.

Polymorphism

- Literal meaning : “Many forms”
- We can use the “many forms” of derived and base classes interchangeably.
- The fact that static and dynamic types of references and pointers can differ is the cornerstone of how C++ supports polymorphism.

Polymorphism.

- C++ supports several kinds of **static (compile-time)** and **dynamic (run-time) polymorphism**.
 - Static Polymorphism
 - Function/Operator Overloading
 - Class/function templates
 - Dynamic polymorphism
 - Polymorphism through inheritance/Virtual member functions

Polymorphism : Example

```
#include <iostream>

class Bird // the "generic" base class
{
public:
    virtual void OutputName() (std::cout << "a bird");
    virtual ~Bird() {}
};

class Swan : public Bird // Swan derives from Bird
{
public:
    void OutputName() (std::cout << "a swan"); // overrides virtual function
};

int main()
{
    Bird* myBird = new Swan; // Declares a pointer to a generic Bird,
    // and sets it pointing to a newly-created Swan.
    myBird->OutputName(); // This will output "a swan", not "a bird".
    delete myBird;
    return 0;
}
```

RTTI: Run time type identification.

C++ has the ability to determine the type of a program's object/variable at runtime.

```
class base {
    virtual ~base(){}
};

class derived : public base {
public:
    virtual ~derived(){}
    int compare (derived &ref);
};

int my_comparison_method_for_generic_sort (base &ref1, base &ref2)
{
    derived & d = dynamic_cast<derived &>(ref1); // rtti used here
    // rtti enables the process to throw a bad_cast exception
    // if the cast is not successful
    return d.compare (dynamic_cast<derived &>(ref2));
}
```

Inheritance Guidelines

- Prefer minimal classes.
 - D&C: Small classes are easier to write, get right, test, use ...
- Prefer composition to inheritance.
- Avoid inheriting from classes that were not designed to be base classes.
- Prefer providing abstract interfaces.

Inheritance

- Differentiate between inheritance of interface and inheritance of implementation
 - Member function interfaces are always inherited.
 - Purpose of pure virtual function is to have derived classes inherit a *function interface* only.
 - Purpose of declaring a simple virtual function is to have derived classes inherit a function interface as well as a default implementation.
 - Purpose of non-virtual function is to have a derived class inherit a function interface as well as a mandatory implementation.

Inheritance

- Never redefine an inherited non-virtual function.
- Never redefine an inherited default parameter value.
 - Virtual functions are dynamically bound but default parameter values are statically bound.

Inheritance and templates.

- Consider the two design problems
 - A stack of objects. Each stack is homogeneous. You might have a stack of ints, strings, ...
 - Classes representing monkeys. You need several different classes representing monkeys (each breed is a little different).
- Sound similar? They result in utterly different software design.

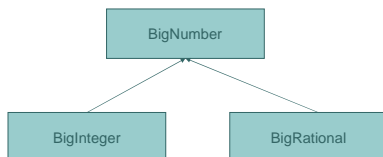
Inheritance and templates.

- With both stacks and monkeys, you are dealing with variety of different types. (objects of type T, monkeys of breed T)
- Question you want to ask yourself:
 - Does the type T affect the behavior of the class?
 - Nope : Use templates
 - Yup: You need virtual functions?

Some real interview questions.

- [What is an explicit constructor?](#)
- [What is a mutable member?](#)
- [Explain the ISA and Has-A class relationships. How would you implement each in a class design?](#)
- [What is a virtual destructor?](#)
- [What is the difference between a copy constructor and an overloaded assignment operator?](#)

Your next assignment.



BigNumber

```
class BigNumber {
...
const int bitsize = 128; // All derived numbers should work upto 128 bits.
virtual void print(); // prints the value of the number to cout

public:
virtual ~BigNumber();
}

Operators overloaded: <<, +, *, >> (Left and right shift)
```

Provide other functions in your interface as needed.
Operator+ and * should work for mixed type arithmetic.



Your library should ...

```
BigInteger i1(7); //default bit length = 128bits
BigRational r2(32,7); //represents 32/7
BigInteger & n1 = i1;
BigRational & n2 = r2;
BigRational n3 = n1 * n2; // n3 should be a Rational
                        // whose value is 32
n1 = (n1 + n3) + n1; // n1 should be an Integer
                        // n1 = 7 + 32 + 7 = 46

cout << n3 << endl;
cout << n2 << endl;
cout << ((n1 + 1) >> 1) << endl; // 4 now
```

Deadline: Nov 20th, 11am

Next homework: BFS on Graphs. Start designing your graph data structure class in C++. You need to submit the spec sheet in the beginning of class on Nov 14th.



Graph libraries

- o LEDA
 - <http://www.mpi-sb.mpg.de/LEDA/MANUAL/graph.html>
- o Boost Graph
 - http://www.boost.org/libs/graph/doc/graph_concepts.html