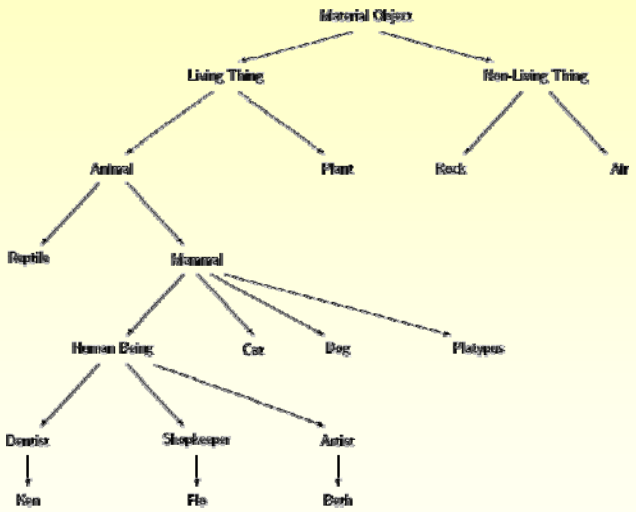
Lecture 5 :
Inheritance & Polymorphism

Acknowledgement : courtesy of Prof. Timothy Budd lecture slides

Object Oriented Programming

- The most basic philosophy of OOP
  - Organizing the system as the interaction of loosely coupled software components.
  - Organizing the classes into a hierarchical structure based on the concept of inheritance
- Inheritance
  - Instance of child class (subclass/derived class) can access both data and behavior(method) of parent class (superclass/base class)

Inheritance : Hierarchical Structure



Inheritance is both Extension and Contraction

- Extension
  - Because the behavior of a child class is larger than the behavior of the parent, the child is an extension of the parent. (larger)
- Contraction
  - Because the child can override behavior to make it fit a specialized (restricted) situation, the child is a contraction of the parent. (smaller)
- Power of OOP
  - Interplay between extension and contraction allows object-oriented systems to take very general tools and specialize them for specific projects

IS-A relationship

- Can check whether two classes have inheritance relationship (general/specialized)
- public inheritance
- Say “A is-a B”. If it ``sounds right'' to your ear. Then A can be made a subclass of B.
  - (ex) A dog is-a mammal (O)
  - (ex) A car is-a engine (X)
- HAS-A relationship
  - Composition (membership) relationship
  - Can be used with private inheritance

Practical meaning of Inheritance

- Member variables and functions in the parent class are part of child class
- Note that **private** aspects of the parent are *part* of the child, but are not accessible within the child class

private, public, protected members

- private
  - accessible only within the class definition
  - but memory is still found in the child class, just not accessible
- public
  - accessible anywhere
- protected
  - accessible within the class definition or within the definition of child classes.
  - **`protected`** menas that we have less encapsulation
  - Use **`protected`** only where it is really necessary

Inheritance : Why Good?

- Reuse of code
  - Methods defined in the parent can be made available to the child without rewriting.
  - Makes it easy to create new abstractions
- Reuse of concept
  - Methods described in the parent can be redefined and overridden in the child
  - Although no code is shared between parent and child, the concept embodied in the definition is shared (ex. draw function)

Benefits of Inheritance

- Software Reuse (Code Sharing)
  - Improved Reliability
  - Improved Maintainability
- Consistency of Interface (abstract class)
- Rapid Prototyping
- Extensibility/Polymorphism
  - New functionality may be easily plugged in without changing existing classes

Cost of Inheritance

- Execution speed (dynamic binding)
- (Program size)
- Program Complexity
- Increased coupling between classes

Constructor in inheritance (order)

- Instantiating a derived class object begins a chain of constructor calls
  - before performing its own tasks,
  - the derived class constructor invokes its direct base class's constructor either explicitly (via a base-class member initializer) or implicitly (calling the base class's default constructor).
- Destructors are called in reverse order

Order of constructor/destructor

```
#include <iostream>
using namespacestd;

class Address {
    public:
          Address() { cout<< "Address's constructor" << endl; }
          ~Address() { cout<< "Address's destructor" << endl; }
};

class Person {
    public:
          Person() { cout<< "Person's constructor" << endl; }
          ~Person() { cout<< "Person's destructor" << endl; }
};

class Student : public Person {
    private: Address address;
    public:
          Student() { cout<< "Student's constructor" << endl; }
          ~Student() { cout<< "Student's destructor" << endl; }
};

Int main() { Student x; }
```

Order of constructor/destructor

1. Person's constructor
2. Address's constructor
3. Student's constructor
4. Student's destructor
5. Address's destructor
6. Person's destructor

```
// This works fine
# include <iostream>
using namespace std;

class B {
   private: int x;
   public:
         B(): x(10) { }
         void displayB() { cout<< "x = " << x << endl; }
};

class D: public B {
   private: int y;
   public:
         D(): y(20) { };// Default Constructor used for B
         void displayD() { cout<< "y = " << y << endl; }
};

void main() {
   D derived;
   derived.displayB();
   derived.displayD();
}
```

```
// This works fine
# include <iostream>
using namespace std;

class B {
   private: int x;
   public:
         B(int x_val): x(x_val) { }
         void displayB() { cout<< "x = " << x << endl; }
};

class D: public B {
   private: int y;
   public:
         D(int x_val,int y_val): B(x_val) , y(y_val) { }
         void displayD() { cout<< "y = " << y << endl; }
};

void main() {
   D derived(10,20);
   derived.displayB();
   derived.displayD();
}
```

```
// compile error . Why?
# include <iostream>
using namespace std;

class B {
   private: int x;
   public:
         B(int x_val): x(x_val) { }
         void displayB() { cout<< "x = " << x << endl; }
};

class D: public B {
   private: int y;
   public:
         D() {}
         void displayD() { cout<< "y = " << y << endl; }
};

void main() {
   D derived;
   derived.displayB();
   derived.displayD();
}
```

Principle of Substitution (Liskov Substitution Principle)

- Property of OOP language
  - a variable declared as the parent type is allowed to hold a value derived from a child type.
  - Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it .
- dynamic binding + virtual function

Dynamic binding

- determining the exact implementation of a request based on both the request (operation) name and the receiving object at the run-time
- When a call to a member function is made using a pointer(or reference) to a base class, the actual member function invoked should be the one implemented in the object that the pointer or reference points to at the time of the call
- This allows us to add new classes to existing systems without modifying the existing code.

C++ virtual function

- virtual function's behavior is defined within an inheriting class (child class) by a function with the same signature
- without virtual
  - member function call depends on the class type of the pointer
- with virtual
  - member function call depends on the actual object to which it points to
- Understand the differences of ex1.cpp ~ ex4.cpp (available on class webpage)
- Understand source codes in sampleProj.zip (available on class webpage)

```
#include <iostream>

class Animal {
  public:
    virtual void eat()
    {
      std::cout << "I eat like a generic Animal.\n";
    }
};

class Wolf : public Animal {
  public:
    void eat() { std::cout << "I eat like a wolf!\n"; }
};

class Fish : public Animal {
  public:
    void eat() { std::cout << "I eat like a fish!\n"; }
};

class OtherAnimal : public Animal {
};

int main()
{
  Animal *anAnimal[4];
  // dynamic binding
  anAnimal[0] = new Animal();
  anAnimal[1] = new Wolf();
  anAnimal[2] = new Fish();
  anAnimal[3] = new OtherAnimal();

  for(int i = 0; i < 4; i++) anAnimal[i]->eat();
  return 0;
}
```

예제

```
Output (with virtual)

I eat like a generic Animal.
I eat like a wolf!
I eat like a fish!
I eat like a generic Animal.
```

```
(without virtual)

I eat like a generic Animal.
I eat like a generic Animal.
I eat like a generic Animal.
I eat like a generic Animal.
```

virtual destructor

- example

```
#include <iostream.h>

class Base
{
       public:
          Base(){ cout<<"Constructor: Base"<<endl;}
          virtual ~Base(){ cout<<"Destructor : Base"<<endl;}
};

class Derived: public Base
{
     //Doing a lot of jobs by extending the functionality
       public:
           Derived(){ cout<<"Constructor: Derived"<<endl;}
           ~Derived(){ cout<<"Destructor : Derived"<<endl;}
};

void main()
{
        Base *Var = new Derived();
        delete Var;
}
```

abstract class

- contains at least one *pure virtual function*.
- Provide interface without implementation
- pure virtual function
  - Declared with “=0”
  - The virtual function is always defined(implemented) in a derived class
- Example

```
class A
{
      public: virtual void f() = 0;
};
```

Properties of abstract class

- No objects of an abstract class can be created
- but pointers and references to an abstract class are still legal.
- An abstract class can only be used as a base class.
- Every derived class of an abstract class must implement the inherited pure virtual functions if objects are to be created from it.

Purpose of abstract class

- provides a uniform interface
  - to deal with a number of different derived classes.
  - Interface reuse
- delegates implementation responsibility to the derived class

example

```
class Vehicle {
    public :
          virtual double accelerate(double) = 0;
          virtual double speed() = 0;
};

class Car : public Vehicle {
    public :
          virtual double accelerate(double);
          virtual double speed();
};

class Bicycle : public Vehicle {
    public :
          virtual double accelerate(double);
          virtual double speed();
};

void full_stop(Vehicle& v) {
    v.accelerate(-v.speed();
}

Vehicle v;  // compile error! It is an abstract class
Car c;
…
full_stop(c);
```

Polymorphism

- many(Poly) + forms (Morph)
- One name (variable, function, class) may have many different meanings

Four different forms of Polymorphims

- Overloading
  - A single function name has several alternative implementations.
  - Typically overloaded function names are distinguished at compile time based on their type signature
- Overriding
  - Different definitions that use the same method name with the same signature
  - A child class redefining a method inherited from a parent class
- Polymorphic variable
  - A variable declared as one type but in fact holds a value of a different type
- Generics (templates)
  - A way of creating general tools or classes by parameterizing on types

Overloading

- Overloading based on scope
  - namespace
- Function overloading
  - Default (optional) parameter
- Operator overloading

Function overloading & default parameter

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

void Func( int one, int two=2, int three=3);
void Func( float fv );

main ()
{
    Func(10, 20, 30);
    Func(10, 20); // Let the last parm default
    Func(10); // Just provide the required parm.
    Func(1.5f);
}

void Func( int one, int two, int three)
{
    cout << "One = " << one << endl;
    cout << "Two = " << two << endl;
    cout << "Three = " << three << endl << endl;
}

void Func ( float fv)
{
    cout << fv << endl;
}
```


C++ : Operator overloading example

```
// operator_overloading.cpp
#include <iostream>
using namespace std;

class Complex {
private:
   double re, im;

public :
   Complex( double r, double i ) : re(r), im(i) {}
   Complex operator+( Complex &other );
   Complex operator-( Complex &other );
   Complex operator*( Complex &other );
   void Display( ) {
                if (im>0) cout << re << "+" << im <<"i"
<< endl;
                else if (im<0) cout << re << im <<"i" << endl;
                else if (im==0) cout << re << endl;
   }
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
   return Complex( re + other.re, im + other.im );
}

Complex Complex::operator-( Complex &other ) {
   return Complex( re - other.re, im - other.im );
}

Complex Complex::operator*( Complex &other ) {
   return Complex( re * other.re - im * other.im
                     , re*other.im + im*other.re );
}
```

```
int main() {
   Complex a = Complex( 1.2, 3.4 );
   Complex b = Complex( 5.6, 7.8 );
   Complex c = Complex( 0.0, 0.0 );

   a.Display();
   b.Display();
        cout << endl;

   c = a + b;
   c.Display();
   c = a - b;
   c.Display();
   c = a * b;
   c.Display();
}
```

```
결과:
1.2+3.4i
5.6+7.8i

6.8+11.2i
-4.4-4.4i
-19.8+28.4i
```


C++ : Operator overloading example (using friend)

```
// operator_overloading.cpp
#include <iostream>
using namespace std;

class Complex {
private:
   double re, im;

public :
   Complex( double r, double i ) : re(r), im(i) {}
   friend Complex operator+(Complex&, Complex&);
   friend Complex operator-(Complex&, Complex&);
   friend Complex operator*(Complex&, Complex&);
   void Display( ) {
                if (im>0) cout << re << "+" << im <<"i" << endl;
                else if (im<0) cout << re << im <<"i" << endl;
                else if (im==0) cout << re << endl;
   }
};

// Operator overloaded using a non-member function
Complex operator+( Complex& i , Complex& other ) {
   return Complex( i.re + other.re, i.im + other.im );
}

Complex operator-( Complex& i , Complex& other ) {
   return Complex( i.re - other.re, i.im - other.im );
}

Complex operator*( Complex& i , Complex& other ) {
   return Complex( i.re * other.re - i.im * other.im
                   , i.re*other.im + i.im*other.re );
}
```

```
int main() {
   Complex a = Complex( 1.2, 3.4 );
   Complex b = Complex( 5.6, 7.8 );
   Complex c = Complex( 0.0, 0.0 );

   a.Display();
   b.Display();
        cout << endl;

   c = a + b;
   c.Display();
   c = a - b;
   c.Display();
   c = a * b;
   c.Display();
}
```

```
결과:
1.2+3.4i
5.6+7.8i

6.8+11.2i
-4.4-4.4i
-19.8+28.4i
```

Operator Functions
As Class Members v.s. As Non-member Functions

- Operator functions
  - Member functions
    - Use **this** keyword to implicitly get argument
    - Gets left operand for binary operators (like **+**)
    - Leftmost object must be of same class as operator
  - Non member functions
    - Need parameters for both operands
    - Can have object of different class than operator
    - Must be a **friend** to access **private** or **protected** data

Overriding

- Overriding only occurs in the context of the parent/child relationship
- The type signatures must match
- Overriding is resolved at run-time, not at compile time

Two possible means for overriding

- Replacement
  - totally and completely replaces the code in the parent class the code in the child class
- Refinement
  - executes the code in the parent class, and adds to it the code in the child class
  - (ex) constructor

Simulating Refinement with Replacement

```
void Parent::example (int a)
{
   cout << "in parent code\n";
}

void Child::example (int a)
{
   Parent::example(12); // do parent code
   cout << "in child code\n"; // then child code
}
```

Polymorphic variable

- (example) `Account* accPtr = new CreditAccount ;`
- A polymorphic Variable is a variable that can reference more than one type of object
- Polymorphic variables derive their power from interaction with inheritance, overriding and substituion.
- When a method is overridden, the code executed will be determined by the current value of a polymorphic variable, not its declared type
- example of common polymorphic variable : this (receiver variable)

Generics (template) : will be covered later

- another approach to software reuse
- basic idea
  - develop code by leaving certain key types unspecified, to be filled in later
- this is like a parameter that is filled with many different values. Here, however, the parameters are types, and not values
- Generics are used both with functions and with classes