Inheritance

Prepared

by

Naveen Choudhary

Inheritance

Base class → Derived class

```
class base
{
};
class derived-class: access base-class
{
/*
access == default for class – private

          == default for structure - public
*/
};
```

- Whatever is the *access* : private members of the base class can not be accessed in the derived class
- If access == public :: Public members of the base class becomes public members of the derived class, Protected members of the base class becomes protected members of the derived class
- if access == private :: All public & protected members of the base class becomes private members in the derived class.

```
class C1{
        private:
        -----
        protected:
        -----
        }; /* protected members can be accessed only
```

by the class members (like private) but they have a greater role in case of inheritance & are treated differently than private, when are used in inheritance.*/

- if access = protected :: public & protected members of the base class becomes protected members of the derived class

Examples

```
#include <iostream>
using namespace std;
class base {
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
  int k;
public:
  derived(int x) { k=x; }
  void showk() { cout << k << "\n"; }
};
int main()
{
  derived ob(3);
  ob.set(1, 2); // access member of base
  ob.show(); // access member of base
  ob.showk(); // uses member of derived class
  return 0;
}
```

```
// This program won't compile.
#include <iostream>
using namespace std;
class base {
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n";}
};

class derived : private base {

// as access specifier is private the public members of
//the base class will become private members of the
//derived class and so can not be accessed from
//outside the class

 int k;
public:
  derived(int x) { k=x; }
  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob(3);
  ob.set(1, 2); // error, can't access set()
  ob.show(); // error, can't access show()
  return 0;
}
```

Examples

```
#include <iostream>
using namespace std;
class base {
protected:
  int i, j; // private to base, but accessible by derived
public:
  void setij(int a, int b) { i=a; j=b; }
  void showij() { cout << i << " " << j << "\n"; }
};
// Inherit base as protected.
class derived : protected base{
  int k;
public:
                // derived may access base's i and j and setij().
  void setk() { setij(10, 12); k = i*j; }
                          // may access showij() here
  void showall() { cout << k << " "; showij(); }
};
int main()
{
  derived ob;
//   ob.setij(2, 3); // illegal, setij() is  protected member of derived

  ob.setk(); // OK, public member of derived
  ob.showall(); // OK, public member of derived
//    ob.showij(); // illegal, showij() is protected  member of derived
  return 0;
}
```

Inheriting Multiple Base Class

```
// An example of multiple base classes.
#include <iostream>
using namespace std;
class base1 {
protected:
  int x;
public:
  void showx() { cout << x << "\n"; }
};
class base2 {
protected:
  int y;
public:
  void showy() {cout << y << "\n";}
};
// Inherit multiple base classes.
class derived: public base1, public base2 {
public:
  void set(int i, int j) { x=i; y=j; } // can access x and y
//as they have become protected members of derived
};
int main()
{
  derived ob;
  ob.set(10, 20); // provided by derived
  ob.showx(); // from base1
  ob.showy(); // from base2
  return 0;
}
```

Constructor, Destructor and Inheritance

- *constructor function are executed in the order of derivation*
- *destructor functions are executed in reverse order of derivation*

```
#include <iostream>
using namespace std;
class base {
public:
  base() { cout << "Constructing base\n"; }
  ~base() { cout << "Destructing base\n"; }
};

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

int main()
{
  derived ob;
  // do nothing but construct and destruct ob
  return 0;
}
```

Output:
```
constructing base
constructing derived
destructing derived
destructing base
```

```
#include <iostream>
using namespace std;
class base1 {
public:
  base1() { cout << "Constructing base1\n"; }
  ~base1() { cout << "Destructing base1\n"; }
};
class base2 {
public:
  base2() { cout << "Constructing base2\n"; }
  ~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
        // note the order of declaration
public:
  derived() { cout << "Constructing derived\n"; }
  ~derived() { cout << "Destructing derived\n"; }
};
int main()
{
  derived ob;
  // construct and destruct ob
  return 0;
}
```

Output:
```
constructing base1
constructing base2
constructing derived
destructing derived
destructing base2
destructing base1
```

Passing parameters to Base-class constructors

```
derived-constrctor (arg_list): base1 (arg_list), base2(arg_list), ... baseN(arg_list)
{
        //body of derived constructor
}
```

```
#include <iostream>
using namespace std;
class base1 {
protected:
  int i;
public:
  base1(int x) { i=x; cout << "Constructing base1\n"; }
  ~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
  int k;
public:
  base2(int x) { k=x; cout << "Constructing base2\n"; }
  ~base2() { cout << "Destructing base1\n"; }
};
class derived: public base1, public base2 {
  int j;
public:
  derived(int x, int y, int z): base1(y), base2(z)
    { j=x; cout << "Constructing derived\n"; }
  ~derived() { cout << "Destructing derived\n"; }
  void show() { cout << i << " " << j << " " << k << "\n"; }
};
```

```
int main()
{
  derived ob(3, 4, 5);
  ob.show(); // displays 4 3 5
  return 0;
}
```

- Passing an argument along to a base class does not preclude its use by the derived class as well

```
class derived :  public base
{               int i;
                public:
// derived use both x & y and then passes them to base
                derived (int x, int y) :base (x,y)
                {
                j = x * y; cout<<"Constructing derived ";
                 }
```

Note:- One final point to keep in mind when passing arguments to base-class constructors, the argument can consist of any expression valid at the time. this includes function calls and variable. this is in keeping with the fact that C++ allows dynamic initialization.

Virtual Base classes

A element of ambiguity can be introdcued into c++ program when multiple base class are inherited

```
// This program contains an error and will not
//compile.

#include <iostream>
using namespace std;
class base {
public:
   int i;
};

// derived1 inherits base.
class derived1 :  public base {
public:
  int j;
};

// derived2 inherits base.
class derived2 : public base {
public:
  int k;
};
```

```
/* derived3 inherits both derived1 and derived2.
   This means that there are two copies of base
   in derived3! */

class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;
  ob.i = 10; // this is ambiguous, which i???
  ob.j = 20;
  ob.k = 30;

 // i ambiguous here, too
  ob.sum = ob.i + ob.j + ob.k;

  // also ambiguous, which i?
  cout << ob.i << " ";
  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;
  return 0;
}
```

Virtual Base classes

There can be 2 Solutions to the problem mentioned on the previous slide

1. Use *scope resolution operator*
2. Use virtual base class

Use of scope resolution operator

```
// This program uses explicit scope resolution to select i.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 inherits base.
class derived1 :  public base {
public:
  int j;
};

// derived2 inherits base.
class derived2 : public base {
public:
  int k;
};
```

```
/* derived3 inherits both derived1 and derived2.
   This means that there are two copies of base
   in derived3! */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;
  ob.derived1::i = 10; // scope resolved, use
                       //derived1's i
  ob.j = 20;
  ob.k = 30;

                // scope resolved
  ob.sum = ob.derived1::i + ob.j + ob.k;

                // also resolved here
  cout << ob.derived1::i << " ";
  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;
  return 0;
}
```

Virtual Base classes

Virtual base class - Only one copy (instead of 2 copies) will be included in derived3, if virtual base class is used

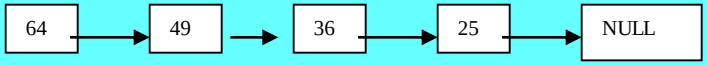
```
// This program uses virtual base classes.
#include <iostream>
using namespace std;
class base {
public:
  int i;
};

// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
  int j;
};

// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
  int k;
};

/* derived3 inherits both derived1 and derived2.
   This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
  int sum;
  };
```

```
int main()
{
  derived3 ob;
  ob.i = 10; // now unambiguous
  ob.j = 20;
  ob.k = 30;
                // unambiguous
  ob.sum = ob.i + ob.j + ob.k;
             // unambiguous
  cout << ob.i << " ";
  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;
  return 0;
}
```

Link List Example

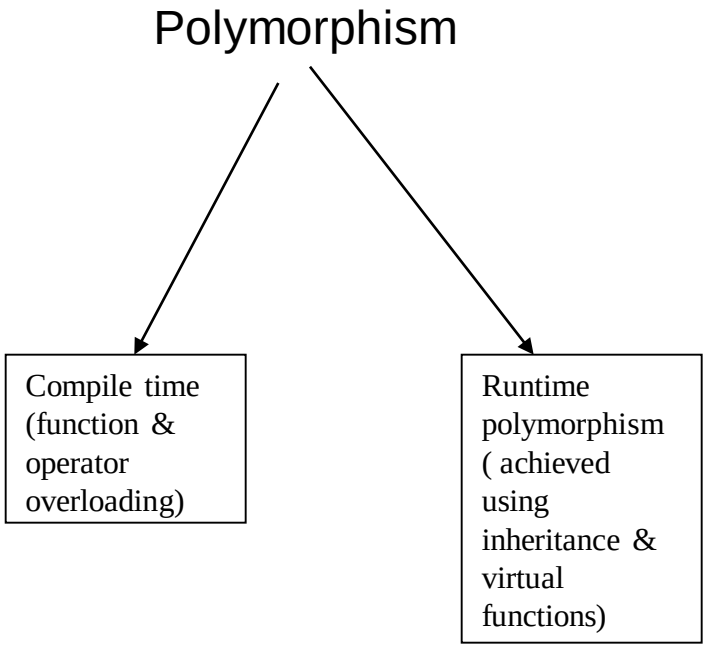
```
class link
{     public:
        int data;
        link *next;
/*a class can't contain an object of the same class
but can contain pointer to object of the base class */
}
class linklist {
                private:
                   link *first ; //pointer to the first link
                public:
                      linklist(){
                                first = NULL;
                                }
                                void additem (int d);
                                void display();
};
void linklist::additem (int d) {
                link *newlink = new link;
                newlink->data = a;
                newlink->next=first;
                first=newlink;
}
void linklist::display() {
                link *current = first;
                while (current != NULL)
                {
                cout<<cuurent->data<<"\n";
                current=current->next;
                }
}
```

```
int main() {
                linklist l;
                l.add(25);
                l.add(36);
                l.add(49);
                l.add(64);
                l.display();
                return 0;
}
```



some questions to be tried in lab ::
How to search & delete an item ?
How to insert an item after a given item?
How to insert an item before a given item?
How to search a given item?

Virtual Function & Polymorphism



Pointers to a derived type (Remember?)

- let B is base class & D be the derived class
- A pointer of type B* (ie say B*ptr) may also point to an object of the type D
- A pointer of type *D (ie say D *ptr1) may not be able to point to an object of type B
- Although we can use a base pointer to point to a derived object, we can access only the members of the derived type that were imported/inherited from the base. that is, we won't be able to access any members added by the derived class {although we can cast a base pointer into derived pointer & gain full access to the entire derived class}

```
class base {
                int i;
        public:
                void set_i(int num) { i=num; }
                int get_i() { return i; }
};
class derived: public base {
                int j;
        public:
                void set_j(int num) { j=num; }
                int get_j() { return j; }
};
```

```
int main()  {
  base *bp;
  derived d;
  bp = &d; // base pointer points to derived object
                // access derived object using base pointer
  bp->set_i(10);
  cout << bp->get_i() << " ";
/* The following won't work. You can't access element of a
derived class using a base class pointer. */

bp->set_j(88);      // error ---  ((derived *)bp)->set_j(88);//ok
cout << bp->get_j();   //error--cout << ((derived *)bp) >get_j();//ok
 return 0;
}
```

Virtual Function & Polymorphism

<u>Virtual function</u> -> Basic concept -> *One interface, multiple methods*

1. **A virtual function is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword *virtual.* when a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs.**

2. **When a base pointer points to a derived object that contain a virtual function, c++ determines which version of that function to call based upon the type of object pointed to by the pointer. And this determination is made at runtime. thus when different objects are pointed to by the pointer; the different versions of the virtual functions are executed.**

   1. **base class - virtual func_name(int)**
      derived class - func_name(int)// method overloading

   2.
```
base_class b1;
base_class *bp1;
derived_class d1;
bp1 = &d1;
bp1->func_name(int) ; // 1. derived version of virtual function is called
                      // 2. decision is made at run-time
```

Examples

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

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

class derived2 : public base {
public:
  void vfunc() {
    cout << "This is derived2's vfunc().\n";
  }
};
```

```
int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;
  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()
  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()
  // point to derived2
  p = &d2;
  p->vfunc(); // access derived2's vfunc()
  return 0;
}
```

Virtual Function Characterstics

1. Virtual function can also be called in a normal manner (ie using dot operator) but then we are not using the benefits of run-time polymorphism

d2.vfunc(); // call derived 2's vfunc()

2. If you change the prototype when you attempt to redefine a virtual function, the function will simply be considered overloaded by the c++ compiler and its virtual nature will be lost.

3. virtual functions must be non static members of the classes of which they are part

4. virtual function can't be a friend function
5. A constructor function can't be virtual but destructor function can be a virtual function

note : constructor can not be virtual because virtual function works depending on the type of object addressed but during construction we don't know what type of object is being constructed

```
class base
{
                public:
1               ~base();
2               //virtual ~base()
                {
                cout << "base destroyed"
                }
};

class derv : public base
{
                public:
                ~derv()
                {
                cout<<"Derv destroyed";
                }
};

int main()
{
                base * pbase = new derv;
                delete pbase;
                return 0;
}
```

O/P→ base destroyed

but if we comment (1) and uncomment (2)

O/P → derv destroyed base destroyed

Note : even if the destructor of base is defined as pure virtual then also we should provide the body for the base destructor as the base destructor will always be called whenever the object is destroyed

Virtual function & Reference

when a virtual function is called through a base class reference, the version of the function executed is determined by the object being referred to at the time of the call

```
/* Here, a base class reference is used to access
   a virtual function. */
#include <iostream>
using namespace std;
class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};
class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};
class derived2 : public base {
public:
  void vfunc() {
    cout << "This is derived2's vfunc().\n";
  }
};
```

```
// Use a base class reference parameter.
void f(base &r) {
  r.vfunc();
}
int main()
{
  base b;
  derived1 d1;
  derived2 d2;
  f(b); // pass a base object to f()
  f(d1); // pass a derived1 object to f()
  f(d2); // pass a derived2 object to f()
  return 0;
}
```

Virtual Attribute Inheritance

The virtual attribute is inherited : no matter how many times a virtual function is inherited, it remains virtual.

```
#include <iostream>
using namespace std;
class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};
class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};
/* derived2 inherits virtual function vfunc()
   from derived1. */
class derived2 : public derived1 {
public:
  // vfunc() is still virtual
  void vfunc() {
    cout << "This is derived2's vfunc().\n";
  }
};
```

```
int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;
  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()
  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()
  // point to derived2
  p = &d2;
  p->vfunc(); // access derived2's vfunc()
  return 0;
}
```

Virtual function are hierarchical

if the derived class fails to override a virtual function of the base class then it not an error & if object of the derived class access the virtual function, the function defined by the base will be used.
In general when a derived class fails to override a virtual function the first redefinition found in reverse order of derivation is used.

```
#include <iostream>
using namespace std;
class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};
class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};
class derived2 : public derived1 {
public:
/* vfunc() not overridden by derived2.
   In this case, since derived2 is derived from
   derived1, derived1's vfunc() is used.
*/
};
```

```
int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;
  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()
  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()
  // point to derived2
  p = &d2;
  p->vfunc(); // use derived1's vfunc()
  return 0;
}
```

Pure Virtual Function

A pure virtual function is a virtual function that has no definition within the base class. To declare pure virtual function, use this general form.

virtual type func_name (arg_list) = 0;

when a virtual function is made pure, any derived class must provide its own definition. if the derived class fails to override the pure virtual function, a compile time error will result.

```
#include <iostream>
using namespace std;
class number {
protected:
  int val;
public:
  void setval(int i) { val = i; }
   virtual void show() = 0; // show() is a pure virtual function
};
class hextype : public number {
public:
  void show() {
    cout << hex << val << "\n";
  }
};
class dectype : public number {
public:
  void show() {
    cout << val << "\n";
  }
};
class octtype : public number {
public:
  void show() {
    cout << oct << val << "\n";
  }
};
int main()
{
  dectype d;
  hextype h;
  octtype o;
  d.setval(20);
  d.show(); // displays 20 - decimal
  h.setval(20);
  h.show(); // displays 14 - hexadecimal
  o.setval(20);
  o.show();  // displays 24 - octal
  return 0;
}
```

Abstract classes

- A class that contains at-least one pure virtual function is said to be abstract. No object of abstract class can be created
- But you can create pointers and reference to an abstract class
- Using abstract class :: one interface - different action depending on slightly different (some what similar) requirements

```
// Virtual function practical example.
#include <iostream>
using namespace std;
class convert {
protected:
  double val1;  // initial value
  double val2;  // converted value
public:
  convert(double i) {
    val1 = i;
  }
  double getconv() { return val2; }
  double getinit() { return val1; }
  virtual void compute() = 0;
};
```

```
// Liters to gallons.
class l_to_g : public convert {
public:
  l_to_g(double i) : convert(i) { }
  void compute() {
    val2 = val1 / 3.7854;
  }
};
// Fahrenheit to Celsius
class f_to_c : public convert {
public:
  f_to_c(double i) : convert(i) { }
  void compute() {
    val2 = (val1-32) / 1.8;
  }
};
```

```
int main()
{
  convert *p;  // pointer to base class
  l_to_g lgob(4);
  f_to_c fcob(70);
// use virtual function mechanism to convert
  p = &lgob;
  cout << p→getinit() << " liters is ";
  p→compute();
  cout << p→getconv() << " gallons\n"; // l_to_g
  p = &fcob;
  cout << p→getinit() << " in Fahrenheit is ";
  p→compute();
  cout << p→getconv() << " Celsius\n"; // f_to_c
  return 0;
}
```