15 Polymorphism & Virtual Functions

- Polymorphism
- Abstract base classes and pure virtual functions
- Virtual functions & (de-)constructors
- Comunicating between two objects
- Operator overloading

The indirect accessings | lymorphism | tion

As explained before, an *ordinary* member function can be rewritten (*overiding*) into another body (the header cannot be changed), you can call them *directly by qualified name*.

As a matter of fact, you can call some functions indirectly when they are defined as virtual, like this:

```
class BaseClass
{ public:
    void MemberFunction()
    { cout<<"BaseClass"<<endl; }
};

class DerivedClass : public BaseClass
{ public:
    void MemberFunction()
    { cout<<"DerivedClass"<<endl; }
};
```

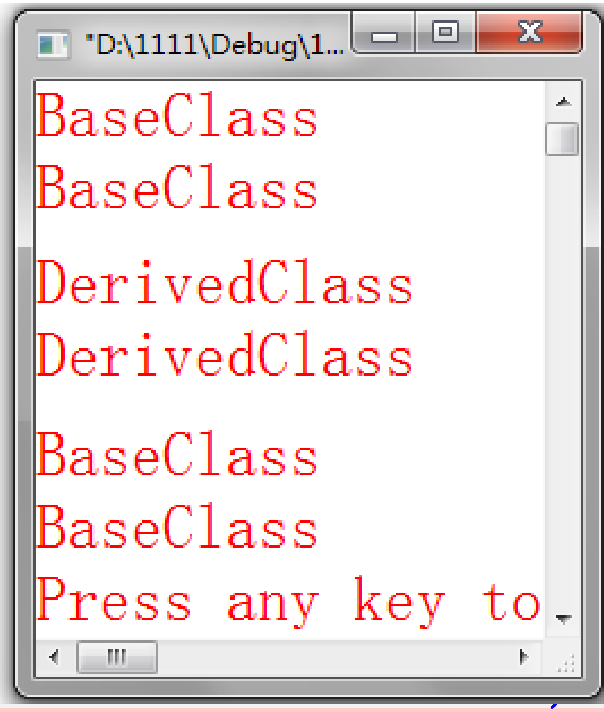
```
BaseClass bc, *pBC=&bc;
DerivedClass dc, *pDC=&dc;

bc.MemberFunction();
pBC->MemberFunction();
cout<<endl;

dc.MemberFunction();
pDC->MemberFunction();

bc=dc;   pBC=pDC;
cout<<endl;
bc.MemberFunction();
pBC->MemberFunction();
```

The indirect accessings tion



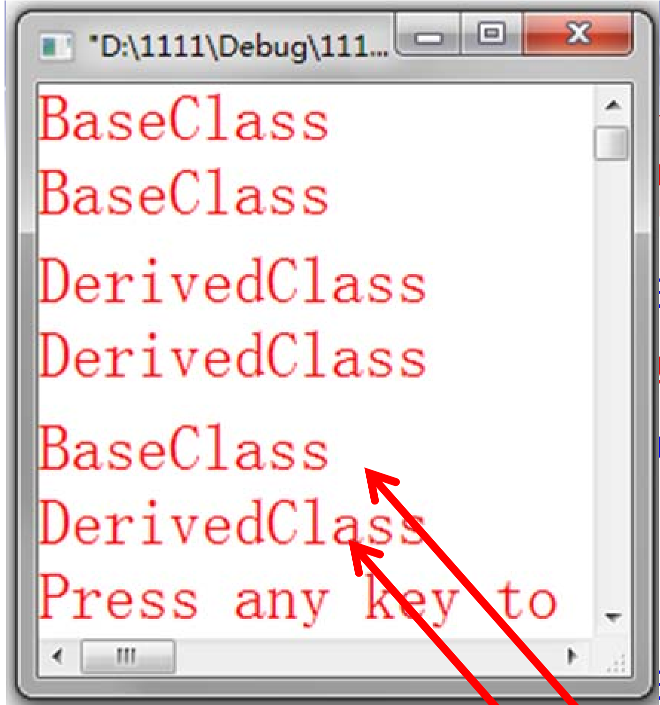
As explained before, an *or* ion can be rewritten (*overiding*) in the header cannot be changed), you *irectly by qualified name*.

As a matter of fact, y functions indirectly when they are d like this:

```
class BaseClass
{ public:
    void MemberFunction()
    { cout<<"BaseClass"<<endl; }
};

class DerivedClass : public BaseClass
{ public:
    void MemberFunction()
    { cout<<"DerivedClass"<<endl; }
};
```

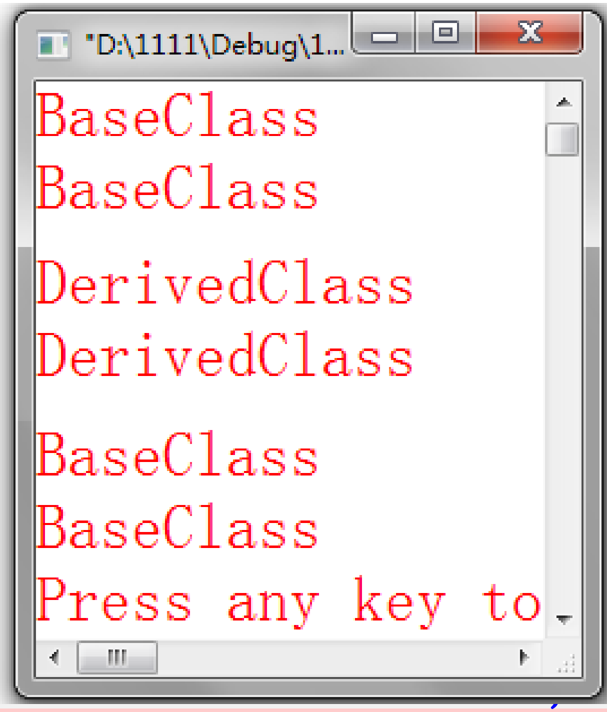
```
BaseClass bc, *pBC=&bc;
DerivedClass dc, *pDC=&dc;
bc.MemberFunction();
pBC->MemberFunction();
cout<<endl;
dc.MemberFunction();
pDC->MemberFunction();
bc=dc;  pBC=pDC;
cout<<endl;
bc.MemberFunction();
pBC->MemberFunction();
```



essings

e, an or
ling) in
d), you

fact, y
ey are d



ymorphism

tion

ion can be
the header
irectly by

functions
like this:

```
class BaseClass
{ public:
    virtual void MemberFunction()
    { cout<<"BaseClass"<<endl; }
};

class DerivedClass : public BaseClass
{ public:
    virtual void MemberFunction()
    { cout<<"DerivedClass"<<endl; }
};
```

```
BaseClass bc, *pBC=&bc;
DerivedClass dc, *pDC=&dc;
bc.MemberFunction();
pBC->MemberFunction();
cout<<endl;
dc.MemberFunction();
pDC->MemberFunction();
bc=dc;  pBC=pDC;
cout<<endl;
bc.MemberFunction();
pBC->MemberFunction();
```

The indirect acc | Polymorphism | tion

It is clear that virtual function can dealing polymorphism.

A virtual function is a member function that you expect to be redefined in subclasses. When you refer to a derived class object using a pointer or *a reference to* the base class, you can call a virtual function for that object and

```
class Employee
{ public:
    virtual void ComputePay() {};
};
class Salesman : public Employee
{ public: void ComputePay() {}};
void PrintPay(Employee* pwe)
{ pwe->ComputePay(); }
int main()
{ Employee aWorker, *empPtr;
  Salesman  aSeller;
empPtr=&aWorker;
empPtr->ComputePay();//Employee::
empPtr=&aSeller;
empPtr->ComputePay();//Salesman::
PrintPay(&aWorker); //Employee::
PrintPay(&aSeller); //Salesman::
return 1; }
```

The indirect acc | Polymorphism | tion

Polymorphism

It is clear that virtual function can dealing polymorphism.

A virtual function is a member function that you expect to be redefined in subclasses. When you refer to a derived class object using a pointer or *a reference to* the base class, you can call a virtual function for that object and

```
class Employee
{ public:
    virtual void ComputePay() {}};
class Salesman : public Employee
{ public: void ComputePay() {} };
void PrintPay(Employee& we)
{ we.ComputePay(); }
int main()
{ Employee aWorker, &rW0=aWorker;
  Salesman  aSeller, &rS0=aSeller;
  rW0=aSeller;
  rW0.ComputePay();//Employee::
```

```
rW0=rS0;
rW0.ComputePay();//Employee::
Employee &rW1=aSeller;
rW1.ComputePay();//Salesman::
rW1=aWorker;      //Take no effect
rW1.ComputePay();//Salesman::
PrintPay(aWorker); //Employee::
PrintPay(aSeller); //Salesman::
PrintPay(rW0); //Employee::
PrintPay(rS0); //Salesman::
return 1; }
```

The indirect acc | Polymorphism | tion

Polymorphism

```
class B {public: virtual func() {}; };

class A1:public B
{public: virtual func() {}; };

class A2:public B
{public: virtual func() {}; };

class A3:public B
{public: virtual func() {}; };

class A4:public B
{public: virtual func() {}; };
```

```
int main()
{
  B *p;
  A1 a1; A2 a2; A3 a3;  A4 a4;

  p=&a1;
  p->func();  //A1::func
  p=&a2;
  p->func();  //A2::func
  p=&a3;
  p->func();  //A3::func
  p=&a4;
  p->func();  //A4::func

  return 1; }
```

Polymorphism

The indirect acc | Polymor | Implementation

```
class Base
{ public:
     //.....
     virtual void Func1() {};
     virtual void Func2() {};
     virtual void Func3() {};     };
class Derived1 : public Base
{ public:
     //.........
     virtual void Func1() {};
     virtual void Func3() {};    };
class Derived2 : public Derived1
{ public:
     // ........
     virtual void Func1() {};    };
int main()
{  Base *pB;
   Base b;
   Derived1  d1;
   Derived2  d2;
}
```

(b).__vfptr	0x00415700 const Base::`vftable'
[0]	0x004110b9 Base::Func1(void)
[1]	0x0041116d Base::Func2(void)
[2]	0x0041114a Base::Func3(void)
d1	{...}
Base	{...}
__vfptr	0x00415714 const Derived1::`vftable'
[0]	0x004110aa Derived1::Func1(void)
[1]	0x0041116d Base::Func2(void)
[2]	0x004110f5 Derived1::Func3(void)
d2	{...}
Derived1	{...}
Base	{...}
__vfptr	0x00415728 const Derived2::`vftable'
[0]	0x004110a5 Derived2::Func1(void)
[1]	0x0041116d Base::Func2(void)
[2]	0x004110f5 Derived1::Func3(void)

pB = &d2
pB -> Func3( )

pB->__vfptr = 0x00415728;
(*(pB->__vfptr + 2))( )

14 Polymorphism & Virtual Functions

- Polymorphism
- Abstract base classes and pure virtual functions
- Virtual functions & (de-)constructors
- Comunicating between two objects
- Operator overloading

Abstract interface

Virtual function and de/con-struction

You can write as following:

```
class Graph
{ public: virtual double Area() = 0; };
```

abstract class

pure virtual function

cannot be initialized

```
class Graph
{ public:
    virtual double Area() = 0;
    virtual Point& OriPos() = 0;
    virtual double Perimeter() = 0;
    virtual double Distance(const Point& p) = 0;
    virtual String& GraphName(const Point& p) = 0;
    virtual char* SetName(const char* pName) = 0;
};
```

Virtual function and de/con-structior

The sequence of calling de/con-structor is important for the class containing virtual function.

We will test them in an experiment in the future

14 Polymorphism & Virtual Functions

- Polymorphism
- Abstract base classes and pure virtual functions
- Virtual functions & (de-)constructors
- Comunicating between two objects
- Operator overloading

Communication

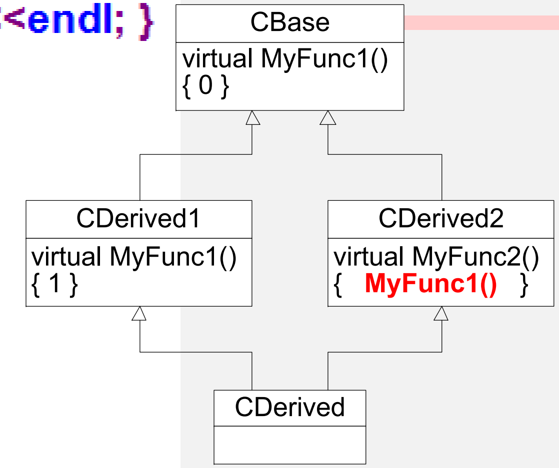
```
class CBase
{ public:
  virtual void MyFunc1() {}
};

class CDerived1 : virtual public CBase
{ public:
  virtual void MyFunc1()  { cout<<"CDerived1::MyFunc1"<<endl; }
};

class CDerived2 : virtual public CBase
{ public:
  virtual void MyFunc2()
  {  cout<<"CDerived2::MyFunc2"<<endl;
     MyFunc1();   }
};

class CDerived : virtual public CDerived1, virtual public CDerived2
{  };

void main()
{ CDerived dObj;
  dObj.MyFunc2(); }
```



14 Polymorphism & Virtual Functions

- Polymorphism
- Abstract base classes and pure virtual functions
- Virtual functions & (de-)constructors
- Comunicating between two objects
- **Operator overloading**

Global operator | perator | Class rator | operators

An operator, e.g. arithmatic operators, generally denotes a kind of operation which can be explained as follows:

z= x+y; <==> z = operator +(x, y);

You can define it as you will

The function can be defined as follow:

<ReturnType> operator@(type1 x, type2 y)
{ …… }

A function can be overloaded, so dose an operator.

Most operators in C++ can be overloaded, such as followings:

- **Unary operators** =====> ! & ~ * + - ++ --, …
- **Binary operators** =====> , != %= -> << <<=, …
- **Function call** =====> *class-type* operator(arg_list)
- **Class member access** =====> *class-type* * operator –>
- **Substripting** =====> *class-type* & operator [ ]

Class member access . cannot be overloaded.

Global operator | …perator | Class …ator | operators

An operator, e.g. arithmatic operators, generally denotes a kind of operation which can be explained as follows:

z= x+y; <==> z = operator +(x, y);

You can define it as you will

The function can be defined as follow:

```
<ReturnType> operator@(type1 x, type2 y)
{ …… }
```

A function can be overloaded, so dose an operator.

```
class CPoint
{ public:
    int m_nX;
    int m_nY;
    CPoint(int x, int y)
    { m_nX=x; m_nY=y;}

    CPoint(void)
    { m_nX=0; m_nY=0;}
};
```

```
CPoint operator + (CPoint cp, int disp)
{  //x轴平移disp
 return CPoint(cp.m_nX+disp, cp.m_nY);
}

int _tmain(void)
{ CPoint cp1(3, 4), cp;
  cp=cp1+3;
  return 1; }
```

Violating encapsulation, data members of CPoint should be private

Global | Friend operator | Class ... | operators

```
class CPoint
{ public:
     int m_nX;
     int m_nY;
     CPoint(int x, int y)
     { m_nX=x; m_nY=y;}
     CPoint(void)
     { m_nX=0; m_nY=0;}
};
```

```
CPoint operator + (CPoint cp, int disp)
{ //x轴平移disp
 return CPoint(cp.m_nX+disp, cp.m_nY);
}

int _tmain(void)
{ CPoint cp1(3, 4), cp;
  cp=cp1+3;
  return 1; }
```

Violating encapsulation, data members of CPoint should be private

```
class CPoint
{ private:
     int m_nX;
     int m_nY;
   public:
     CPoint(int x, int y)
     { m_nX=x; m_nY=y;}
     CPoint(void)
     { m_nX=0; m_nY=0;}
     int GetX() { return m_nX; }
     int GetY() { return m_nY; }   };
```

Meeting encapsulation with lower efficiency

```
CPoint operator + (CPoint cp, int disp)
{ //x轴平移disp
 return CPoint(cp.GetX()+disp, cp.GetY());
}

int _tmain(void)
{ CPoint cp1(3, 4), cp;
  cp=cp1+3;
  return 1;  }
```

public access function

Global | Friend operator | Class | operators

```
class CPoint
{ private:
      int m_nX;
      int m_nY;
   public:
      CPoint(int x, int y)
      { m_nX=x; m_nY=y;}

      CPoint(void)
      { m_nX=0; m_nY=0;}

      int GetX() { return m_nX; }
      int GetY() { return m_nY; }   };
```

Meeting encapsulation with lower efficiency

```
CPoint operator + (CPoint cp, int disp)
{  //x轴平移disp
  return CPoint(cp.GetX()+disp, cp.GetY());
}

int _tmain(void)
{ CPoint cp1(3, 4), cp;
  cp=cp1+3;
  return 1;  }
```

public access function

```
class CPoint
{ private:
     int m_nX;
     int m_nY;
  public:
    CPoint(int x, int y) { m_nX=x; m_nY=y;}
    CPoint(void) { m_nX=0; m_nY=0;}

friend CPoint operator + (CPoint cp, int disp);

};
```

Alternative

```
CPoint operator + (CPoint cp, int disp)
{  return CPoint(cp.m_nX+disp, cp.m_nY);  }

int _tmain(void)
{ CPoint cp1(3, 4), cp;
  cp=cp1+3;
  return 1;  }
```

A operator is explained before as followings:

z = x+y; <====> z = operator + (x, y);   In C style

It can be also explained as followings:

z = x+y; <====> z = x.operator + (y);   In C++ style

The first argument must be a object instance

```
class CPoint
{ private:
    int m_nX;
    int m_nY;
  public:
    CPoint(int x, int y) { m_nX=x; m_nY=y;}
    CPoint(void) { m_nX=0; m_nY=0;}

    CPoint operator + (int disp)
    {  return CPoint(m_nX+disp, m_nY);  }
};
```

```
int _tmain(void)
{ CPoint cp1(3, 4), cp;
  cp=cp1+3;
  return 1;  }
```

Equivalent to cp1.+(3)

Overloading operators ++/--

Besides global/friend/class for operators, you should remember:

- **++/-- operand**, operand value after operating = operand +1/-1
  expresson value = operand +1/-1
- **oprand ++/--**, operand value after operating = operand +1/-1
  expresson value = operand

Global operator

Pre operator: **RetType operator @(ArgType ar)**

Post operator: **RetType operator @(ArgType ar, int)**

Class operator

Pre operator: **RetType operator @( )**

Post operator: **RetType operator @(int)**

Overloading operators ++/--

Besides global/friend/class for operators, you should remember:

- **++/-- operand**, operand value after operating = operand +1/-1
  expresson value = operand +1/-1
- **oprand ++/--**, operand value after operating = operand +1/-1

```
class CPoint
{ private:   int m_nX;
             int m_nY;
  public:
   CPoint(int x, int y) { m_nX=x; m_nY=y;}
   CPoint(void) { m_nX=0; m_nY=0;}
   CPoint operator ++ () // Pre decreament
   { m_nX += 1;  m_nY += 1;
     return CPoint(m_nX, m_nY);  }
   CPoint operator -- (int) // Post decreament
   { CPoint p(*this);
     m_nX -= 1;  m_nY -= 1;
     return p;  }  };
```

```
int _tmain(void)
{ CPoint cp(0, 0), cp1(1, 2), point;
  point = ++cp;
  point = cp1--;
  return 1;  }
```

Watch 1

Name	Value
cp	{x=??? y=???}
m_nX	1
m_nY	1
point	{x=??? y=???}
m_nX	1
m_nY	1

Global | Friend ope | | **Specific operators**

The assignment operator (=) is, strictly speaking, a binary operator. Its declaration is identical to any other binary operator, with the following exceptions:

- **It must be a nonstatic member function.** **Note： = must be a member**
- **It is not inherited by derived classes.**
- **A default '=' function will be generated as below by the compiler if needed**
  Memberwise assignment operator declared in the form:

  type& type :: operator = ([const | volatile] type &) **maybe unlike this**

```
class Point
{ private: int m_nX;
           int m_nY;
  public:  Point(int x, int y)
           { m_nX=x; m_nY=y;}
        friend class Complex;
};
int _tmain(void)
{ Point pnt(1, 2), pnt1(5, 6);
  //Complex cplx = pnt; // Error
  Complex cplx(0, 0), cplx1(5, 6);;
  cplx = pnt;
  cplx = cplx1;
  return 1;  }
```

Your attention please !

```
class Complex
{  private:  int m_nR;    int m_nI;
   public:
     Complex(int r, int i) { m_nR=r; m_nI=i; }
     Complex& operator = (const Point& pnt)
     { m_nR=pnt.m_nX;  m_nI=pnt.m_nY;
       return *this; }
     Complex& operator = (Complex& cplx)
     { //Necessary for containing pointer member
       m_nR=cplx.m_nR;   m_nI=cplx.m_nI;
       return *this;  }
} ;
```

The assignment operator (=) is, strictly speaking, a binary operator. Its declaration is identical to any other binary operator, with the following exceptions:

- It must be a nonstatic member function.
- It is not inherited by derived classes.
- A default '=' function will be generated as below by the compiler if needed

  Memberwise assignment operator declared in the form:

  type& type :: operator = ([const | volatile] type &)

```
class MyClass
{ private: int m_nN;
  public:
    MyClass(int n=0) { m_nN=n; }
 /* MyClass operator = (int n)
    { this->m_nN=3;
      return *this;
    } */
};

int _tmain(int argc, _TCHAR* argv[])
{ MyClass c=1;
  c=MyClass(2);
  c=3;
  return 0;
}
```

If no operator '=' is defined by program, some suitable constructor might function as '=' .

If a suitable '=' is defined, it will be called all at a high priority.

Note that, a program defined '=' might finish type conversion at the same time as finishing the assigning operation.

The conversion is from the right to the left.

Global | Friend ope | | Specific operators

Type Conversions. It depend on the specified operator and the type of the operand or operators.Type conversion are performed in the following cases:

- When a value of one type is assigned to a variable of a different type or an operator converts the type of its operand or operands before performing an operation.
- When a value of one type is explicitly cast to a different type.
- When a value is passed as an arguments to a function or when a type is returned from a function

Type cast of an object:

operator type ( );

```
class MyClass
{ private: int m_nN;
  public:
    MyClass(int n=0) { m_nN=n; }
    MyClass operator = (int n)
    { this->m_nN=3; return *this; }
    operator int( ) { return m_nN; }
};
int _tmain(int argc, _TCHAR* argv[])
{ MyClass c=1;
  c=MyClass(2);
  c=3;
  c=c+4;
  return 0; }
```

Please pay more attention to the coordination among the *type conversion operator*, the *constructor* and the *assignment operator* !!!!

There are 2 ways to do this at least

Global | Friend ope | Cl... or | Specific operators

Type Conversions. It depend on the specified operator and the type of the operand or operators.Type conversion are performed in the following cases:

- When a value of one type is assigned to a variable of a different type or an operator converts the type of its operand or operands before performing an operation.
- When a value of one type is explicitly cast to a different type.
- When a value is passed as an arguments to a function or when a type is returned from a function

The other two commonly met operators are ‘<<’ and ‘>>’, you are expected to learn their overload by your self.

THE END