2023
面向对象程序设计

第六讲：　多态性与虚函数

李际军　　lijijun@cs.zju.edu.cn

目录 /Contents

01 多态性

02 运算符重载

03 不同类型数据间的转换

04 虚函数

05 纯虚函数与抽象类

学习目标

（1）掌握多态性的概念。
（2）掌握重载运算符的定义方法。
（3）掌握运算符重载为成员函数的定义方法。
（4）掌握运算符重载为友元函数的方法
（5）掌握不同类型数据间的转换方法。
（6）掌握虚函数的定义和使用方法。
（7）掌握纯虚函数和抽象类的定义。

6.1 多态性

多态性是面向对象程序设计的一个重要特征。顾名思义，多态的意思是一个事物有多种状态。通常我们希望所设计的类具有共同的风格。例如，在不同的类中具有相似功能的函数具有相同的名字、具有相同的参数类型、参数的顺序也相同。这种统一性帮助我们记忆，且有助于新类的设计。在新类的设计中只需要添加相同的数据成员并改写相应的成员函数。不同类的对象调用自己的函数成员，这就是多态性。

6.1.1 多态的类型

- 多态性的实现方式有 4 种：**重载多态、强制多态、类型参数化多态和包含多态**。
- **重载多态**：前面介绍过的函数重载和本章将要介绍的运算符重载都属于重载多态；
- **强制多态**：就是将一个变量的类型加以强制转换来满足某种操作要求，本章介绍的强制类型转换就属于强制多态；
- **类型参数化多态**：是指当 1 个函数（或类）对若干个类型参数操作时，这些类型具有某些公共的语义特性， C++ 中的类模板是实现类型参数化多态的工具，关于类模板的相关内容将在本书中的第 7 章进行详细介绍；
- **包含多态**：类族中定义于不同类中的同名成员函数的多态行为，主要是继承过程中通过虚函数来实现，本章介绍的虚函数属于包含多态。

6.1.2 静态关联与动态关联

- 关联（ binding ）是指捆绑或连接的意思，即把两样东西捆绑在一起。就是确定调用的具体对象的过程，一般来说，关联指把一个标识符和一个存储地址联系起来。
- 关联分为**静态关联**与**动态关联**。如果在编译程序时就能确定具体的调用对象，称为**静态关联**；如果在编译程序时还不能确定具体的调用对象，只有在程序运行过程中才能确定具体调用对象，称为**动态关联**。由于静态关联是在程序运行前进行关联的，所以又称为**早期关联**，而动态关联是在程序运行中进行关联的，也叫作**滞后关联**。

6.1.2 静态关联与动态关联（续）

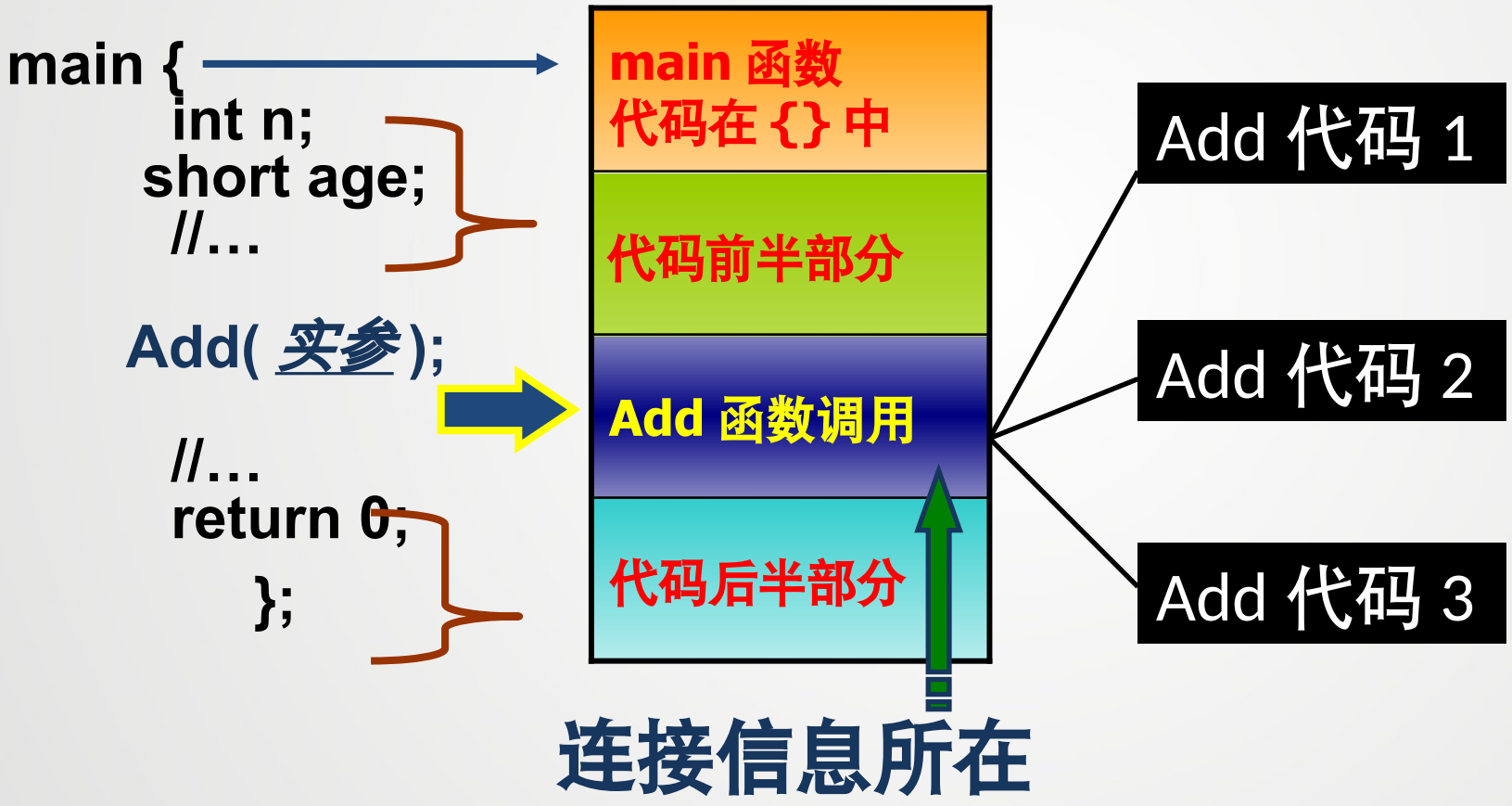
- 静态多态在程序编译时系统就能决定调用的哪个函数，因此又称为编译时的多态性。静态多态性是通过函数的重载实现的，以前学过的函数重载和本章将要学习的运算符重载实现的多态性属于静态多态性。
- 动态多态是在程序运行过程中才动态地确定操作的对象，在运行阶段确定关联关系，又称运行时的多态性。在运行阶段，基类指针变量先指向某一个类对象，然后通过此指针变量调用该对象中的虚函数。此时调用哪一个对象的虚函数无疑是确定的，只是在运行阶段才把虚函数和对象绑定在一起。

6.1.3 多态性的实现原理

- 多态性实现和编译连接、执行密切相关；
- C++ 源程序中函数的经过编译后形成不同的目标代码块（OBJ 等），这些代码和库要进行连接形成最终代码（EXE 等）；
- 目标代码中函数调用处含有连接信息；
- 连接信息含有被调用函数的地址，若地址固定就成为静态地址值；若地址不固定就是指针变量（函数指针）。

6.1.3 多态性的实现原理

- 图例说明多态的实现



6.1.3 多态性的实现原理

- 编译后连接信息中函数地址值固定不变，在此基础上实现的多态称为**静态多态**；
- C++ 中，静态多态表现为函数重载和运算符重载（特殊的函数重载）；
- 编译后连接信息中函数地址值用指针指向，在此基础上实现的多态称为动态多态；
- C++ 中，动态多态是通过虚函数实现的。

6.2 运算符重载

6.2 运算符重载

运算符重载是指对已有的运算符赋予它新的含义，是通过运算重载函数来实现的，本质上也是属于函数重载，是 C++ 实现静态多态的一重要手段。

- 运算符重载是使同一个运算符作用于不同类型的数据时具有不同的行为。
- 运算符重载是实质上将运算对象转化为运算函数的实参，并根据实参的类型来确定重载的运算函数。
- 运算符重载和类型重载是多态的另外两种表现形式。

6.2.1 什么是运算符重载

实际上，我们已经不知不觉之中使用了运算符重载。例如，大家都习惯于用加法运算符“+”对整数、单精度和双精度数进行加法运算，如 5+8 ，5.8+3.67 ，这是因为 C++ 系统已经对于整型数、单精度和双精度数重载了 + 运算符。虽然计算机对于整型数和浮点数的相加过程很不相同，但用户使用这个运算符时完全相同。

6.2.2 运算符重载的方法

运算符重载实质上是函数的重载。

运算符重载函数的一般格式如下：

函数类型 operator 运算符名称（形参表列）

{对运算符的重载处理}

例如，若要对用户定义的类型 Complex 重载实现加法操作，重载函数原型如下：

```
Complex operator+(Complex &a, Complex &b);
```

6.2.2 运算符重载的方法（续）

在定义了重载运算符的函数后，可以说，函数 operator+ 重载了运算符 + 。在执行复数相加的表达式 c1+c2 时，系统就会调用 operator+ 函数，把 c1 和 c2 作为实参，与形参 a 、 b 进行虚实结合，执行函数 operator+（ a ， b ）。

【例 6-1 】 将运算符“ +” 重载为适用于复数加法。这里重载函数作为类的友元函数。

```
#include <iostream>
using namespace std;
class Complex{
public:
        Complex( ){real=0;imag=0;}
        Complex(double r,double i){real=r;imag=i;}
        Complex operator+(Complex &c2);          // 声明重载运算符的函数
        void display( );
 private:
        double real;
        double imag;
};
Complex Complex::operator+(Complex &c2)       // 定义重载运算符的函数
{
        Complex c;
        c.real=real+c2.real;
        c.imag=imag+c2.imag;
        return c;
}
```

```
void Complex::display( )
{ cout<<″(″<<real<<″,″<<imag<<″i)″<<endl;}

int main( ){
        Complex c1(3,4),c2(5,-10),c3;
        c3=c1+c2;                              // 运算符 + 用于复数运算
        cout<<″c1=″;          c1.display( );
        cout<<″c2=″;          c2.display( );
        cout<<″c1+c2=″;       c3.display( );
        return 0;
}
```

运行结果：

```
c1=(3+4i)
c2=(5-10i)
c1+c2=(8,-6i)
```

6.2.3 重载运算符的规则

（1）不允许用户自己定义新的运算符。

（2）并不是所有的运算符都可以进行重载。

（3）操作符所允许的操作数的个数、优先级和结合性不能变。

（4）重载运算符的函数不能有默认参数，否则就改变了运算符参数的个数，与前面第（3）点产生矛盾。

（5）重载的操作符必须有一个用户定义的类型作为操作数。

```
int operator+(int,int);// error:  不能对内置类型重载 +
Vector operator+(const Vector&, const Vector &) ;  // ok
```

（6）用户定义的类型都自动拥有“=”、“&”、“，”运算符，除非有特殊需要，一般不必重载这3个运算符。

（7）建议：重载的操作符的意义要和传统的意义相符。

（8）建议：不要轻易地用重载。

6.2.3 重载运算符的规则

不能重载的运算符只有 5 个：

. （成员访问运算符）

.* （成员指针访问运算符）

:: （域运算符）

sizeof （长度运算符）

?: （条件运算符）

6.2.4 运算符重载为成员函数和友元函数

运算符的重载形式有两种:

(1) 重载为类的成员函数;

(2) 重载为类的友元函数。

```
friend Complex operator+(Complex & a, Complex & b);
Complex operator+( Complex & b);
```

如果将运算符重载作为成员函数，由于它可以通过 this 指针自由访问本类的数据成员，因此可以少写一个函数的参数。但必须要求运算表达式第一个参数(即运算符左侧的操作数)是一个类对象，因为必须通过类的对

【例 6-2 】 将运算符“ +” 重载为适用于复数加法。重载函数作为类的成员函数。

```
#include <iostream>
using namespace std;
class Complex
{
public:
        Complex() {real=0;imag=0;}
        Complex(double r, double i) {real=r; imag=i; }
        //friend Complex operator+(Complex &a, Complex &b);
        Complex operator+( Complex &b);
     void display();
private:
    double real;
    double imag;
};
//Complex operator+(Complex &a, Complex &b)
//{return Complex(a.real+b.real, a.imag+b.imag);}
```

【例 6-2 】 将运算符“ +” 重载为适用于复数加法。重载函数作为类的成员函数。（续）

```
Complex Complex::operator+( Complex &b)
{return Complex(real+b.real, imag+b.imag);}

void Complex::display()
{cout<<"("<<real<<","<<imag<<"i)"<<endl;}

int main()
{
        Complex c1(3,4), c2(5,-10),c3;
    c3=c1+c2;// 执行 c1.operator+(c2);
    cout<<"c1="; c1.display();
    cout<<"c2="; c2.display();
    cout<<"c1+c2="; c3.display();
    getchar();
}
```

例 6-2 与例 6-1 的区别是重载运算符作为成员函数。运算符重载时，什么时候应该用成员函数，什么时候应该用友元函数，两者的区别是什么?

如果将运算符重载作为成员函数，由于它可以通过 this 指针自由访问本类的数据成员，因此可以少写一个函数的参数。但必须要求运算表达式第一个参数（即运算符左侧的操作数）是一个类对象，因为必须通过类的对象去调用该类的成员函数，而且重载函数的返回值与该对象类型相同，只有运算符重载函数返回值与该对象同类型，运算结果才有意义。本例是建立一个 Complex 类，重载函数只有一个参数 b ，相对于友元方式少一个参数 a ，函数的返回值是 Complex 类型的对象。本例将友元函数的实现方式用注释的方式写在代码中，方便读者理解。大部分运算符即可重载为成员函数也可以重载为友元函数。

例 6-2 与例 6-1 的区别是重载运算符作为成员函数。运算符重载时，什么时候应该用成员函数，什么时候应该用友元函数，两者的区别是什么？（续）

C++ 规定，当重载以下的运算符时，必须重载为某个类的成员函数：

= 、 [ ] 、（ ）,->

当重载以下的运算符时，必须是普通函数或友元函数，不能为成员函数：

>> 、 <<

一般将双目运算符重载为友元函数，单目运算符重载为成员函数。

由于友元的使用会破坏类的封装性，因此从原则上说，要尽量将运算符函数作为成员函数。

6.2.5 重载双目运算符

双目运算符（或称二元运算符）是 C++ 中最常用的运算符。双目运算符有两个操作数，通常在运算符的左右两侧，如 3+5，a=b，i<10 等。下面举一个例子说明用友元函数的方法重载双目运算符的应用。

【例 6-3】定义一个字符串类 String，用来存放不定长的字符串，重载运算符“==”，“<”和“>”，用于两个字符串的等于、小于和大于的比较运算。

```
#include <iostream>
 using namespace std;
 class String{
public:
    String() {p=NULL;}
    String(char *str);
    void display();
private:
    char *p;
};
String::String(char *str)
{ p=str;}
void String::display()
{ cout<<p;}
```

```
int main()
{
   String string1("Hello"), string2("Book");
   string1.display();
   cout<<endl;
   string2.display();
   getchar();
   return 0;
}
```

这是一个可运行的简单的框架程序。

在定义对象 string1 时给出字符串“ Hello” 作为参数，它的起始地址传递给构造函数的形参指针 str 。在构造函数中，使 p 指向“ Hello” 。执行 main 函数中的 string1.display() 时，输出 p 指向的字符串“ Hello” 。在定义对象 string2 时给出字符串“ Book” 作为实参，同样，执行 main 函数的 string2.display() 时，就输出 p 指向的字符串“ Book” 。

有了这个基础后，再增加其他必要的内容。现在增加对运算符重载的部分，为便于编写和调试，先重载一个运算符“ >” ，程序如下。

在 String 类中声明一个友元函数：

```
friend bool operator >(String &string1, String &string2);
```

在类外定义“ >” 运算符的重载函数：

```
bool operator >(String &string1, String &string2)
{ if(strcmp(string1.p, string2.p)>0)
    return true;
 else return false;
}
```

再修改主函数：

```
int main()
{  String string1(“Hello”), string2(“Book”);
   cout<<(string1>string2)<<endl;
}
```

6.2.6 重载单目运算符

单目运算符只有一个操作数，如 !a,-b,&c,*p, 还有最常用的 ++i 和 --i 等。重载单目运算符的方法与重载双目运算符的方法是类似的。但由于单目运算符只有一个操作数，因此运算符重载函数只有一个参数，如果运算符重载函数作为成员函数，则还可省略此参数。

下面以自增运算符”++”为例，介绍单目运算符的重载。

6.2.6 重载单目运算符

【例 6-4 】有一个 time 类，包含数据成员 minute(分) 和 sec(秒)，模似秒表，每次走一秒，满 60 秒进一分钟，此时秒又从 0 开始算。要求输出分和秒的值。

```
#include <iostream>
using namespace std;
class Time
{ public:
    Time() {minute=0; sec=0;}
    Time(int m, int s):minute(m), sec(s) { }
    Time operator++();
void display(){cout<<minute<<":"<<sec<<endl;}
 private:
  int minute;
  int sec;
 };
```

6.2.6 重载单目运算符

```
Time Time::operator++()
{ if(++sec>=60)
    { sec-=60;
     ++minute;}
     return *this;
}
int main()
{ Time time1(34,0);
 for(int i=0; i<61;i++)
   {++time1;
    time1.display(); }
  getchar();
  return 0;
 }
```

6.2.6 重载单目运算符

【例 6-5 】 在例 6-4 的基础上增加对后置运算符的重载。修改后的程序如下。

在类的定义中增加函数成员：

```
Time operator++(int);
```

定义后置自增运算符“ ++” 重载函数：

```
Time time::operator++(int)
{ Time temp(*this);
 sec++;
 if(sec>=60)
    {sec-=60;
    ++minute;}
  return temp;
}
```

6.2.6 重载单目运算符

主函数：

```
int main()
{
    Time time1(34,59), time2;
    cout<<"Time1:";
    time1.display();
    ++time1;
    cout<<"++time1";
    time1.display();
    time2=time1++;
    cout<<"time1++:";
    time1.display();
    cout<<"time2:";
    time2.display();
}
```

6.2.6 重载单目运算符

【例 6-5】 在例 6-4 的基础上增加对后置运算符的重载。修改后的程序如下。(续)

在例 6-5 中，重载后置自增运算符时，多了一个 int 型的参数，增加这个参数只是为了与前置自增运算符重载函数有所区别，此外没有任何作用，在定义函数时也不必使用此参数，因此可省写参数名，只需要在括号中写参数类型 int 即可。编译系统在遇到重载后置自增运算符时，会自动调用此函数。

6.2.7 重载流插入运算符和流提取运算符

用户自己定义的类型的数据，是不能直接用“ <<” 和“ >>” 来输出和输入的。如果想用它们输出和输入自己定义的类型的数据，必须在自己定义的类中对这两个运算符进行重载。

对“ <<” 和“ >>” 重载的函数形式如下：

istream &operator>>(istream &，自定义类 &);

ostream &operator<<(ostream &，自定义类 &);

6.2.7 重载流插入运算符和流提取运算符

```
ostream& operator<<(ostream & output, Complex &c)
{ output<<"("<<c.real<<"+"<<c.imag<<"i)";
 return output;
}
istream& operator>>(istream & input, Complex &c)
{ cout<<"input real part and imaginary part of complex
    number:";
 input>>c.real>>c.imag;
 return input;
}
```

6.2.7 重载流插入运算符和流提取运算符

【例 6-6 】 用友元函数重载流插入运算符“ <<” 和流提取运算符“ >>” 。

```
class Complex
 { public:
    friend ostream& operator<<(ostream &, Complex &);
    friend istream& operator>>(istream &, Complex &);
  private:
    double real;
    double imag;
  };
```

【例 6-6 】 用友元函数重载流插入运算符“ <<” 和流提取运算符“ >>” 。 （续）

```
ostream& operator<<(ostream & output, Complex &c)
{ output<<"("<<c.real<<"+"<<c.imag<<"i)";
 return output;
}
istream& operator>>(istream & input, Complex &c)
{ cout<<"input real part and imaginary part of complex
number:";
 input>>c.real>>c.imag;
 return input;
}
```

6.2.7 重载流插入运算符和流提取运算符

【例 6-6】 用友元函数重载流插入运算符“ <<” 和流提取运算符“ >>”(续)

```
int main()
{
    Complex c1,c2;
    cin>>c1>>c2;
  cout<<"c1="<<c1<<endl;
  cout<<"c2="<<c2<<endl;
  cin.clear();// 以下几行是为了程序运行结束时不自动关闭窗口
  cout << "Please enter a character to exit\n";
  char ch;
  cin >> ch;
  return 0;
}
```

6.2.8 下标运算符 [] 的重载

- 在 C++ 中，在重载下标运算符 [] 时，认为它是一个双目运算符，例如 X[Y]
- 对于下标运算符重载定义只能使用成员函数，其形式如下：

```
返回类型   类名 ::operator[]( 形参 )
{
  // 函数体
}
```

6.2.8 下标运算符 [] 的重载

// **Program Example 【例 6-7 】**

```
// Program to demonstrate the overloading of index operator [].
#include <iostream>
using namespace std ;

class int_array // A smart integer array.
{
 public:
              int_array( int number_of_elements = 10 ) ;
             int_array( int_array const &array ) ;
             ~int_array() ;
             int_array const& operator=( int_array const &array ) ;
             int &operator[]( int index ) ;
 private:
              int number_of_elements ;
             int* data ;
             void check_index( int index) const;
             void copy_array( int_array const &array ) ;
} ;
```

6.2.8 下标运算符 [] 的重载


```
// Constructor.
int_array::int_array( int n )
{
        if ( n < 1 )
        {
                cerr << "number of elements cannot be " << n
                << ", must be >= 1" << endl ;
                exit( 1 ) ;
        }
        number_of_elements = n ;
        data = new int [number_of_elements] ;
        for (int i = 0 ; i < number_of_elements ; i++ )
                data[ i ] = 0 ;
}
```

6.2.8 下标运算符 [] 的重载


```
// Copy constructor.
int_array::int_array(int_array const &array)
{
        copy_array(array) ;
}

// Destructor.
int_array::~int_array()
{
        delete[] data ;
}
```

6.2.8 下标运算符 [] 的重载


```
// Assignment operator.
int_array const& int_array::operator=( int_array const &array )
{
        if ( this != &array ) // Avoid self assignment.
        {
                delete[] data ;
                copy_array( array ) ;
        }
        return *this ;
}
```

6.2.8 下标运算符 [] 的重载


```
// Overloaded index operator.
int& int_array::operator[]( int index )
{
        check_index( index ) ;
        return data[ index ] ;
}

void int_array::copy_array( int_array const &array )
{
        number_of_elements = array.number_of_elements ;
        data = new int [ number_of_elements ] ;
        for ( int i = 0 ; i < number_of_elements ; i++ )
                data[ i ] = array.data[ i ] ;
}
```

- 重载下标运算符 [ ] 只是在返回数组元素前检查下标值。

6.2.8 下标运算符 [] 的重载


```
main()
{
        int_array a( 15 ) ; // An array of 15 integers.
        int i ;

        // Display the contents of the array.
        for ( i = 0 ; i < 15 ; i++ )
                cout << a[i] << ' ' ;
        cout << endl ;

        // Assign some values to the elements of the array.
        for ( i = 0 ; i < 15 ; i++ )
                a[i] = i * 10 ;

        // Display the new contents of the array.
        for ( i = 0 ; i < 15 ; i++ )
                cout << a[i] << ' ' ;
        cout << endl ;

  exit( 0 ) ;
}
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140
```

6.2.8 下标运算符 [] 的重载


```
void int_array::check_index( int index ) const
{
  if ( index < 0 || index >= number_of_elements )
  {
    cerr << "invalid index " << index
         << ", range is 0 to " << number_of_elements - 1 << endl ;
    exit( 1 ) ;
  }
}
```

- 如果下标值无效，显示错误信息并终止程序。

6.2.9 函数调用运算符（）的重载

- 在 C++ 中，在重载函数调用运算符 () 时，认为它是一个双目运算符，例如 X(Y)
- 对于函数调用运算符重载定义只能使用成员函数，其形式如下：

返回类型　类名 :: operator (形参表)

```
{
    // 函数体
}
```

一个典型的例子

【例 6-8 】 先建立一个 Point(点) 类，包含数据成员 x,y(坐标点) 。以它为基类，派生出一个 Circle(圆) 类，增加数据成员 r(半径) ，再以 Circle 类为直接基类，派生出一个 Cylinder(圆柱体) 类，再增加数据成员 h(高) 。要求编写程序，重载运算符“ <<” 和“ >>” ，使之能用于输出以上类对象。

(1) 声明基类 Point 类

```
#include <iostream>

// 声明类 Point
class Point
{
public:
    Point(float x=0,float y=0);// 有默认参数的构造函数
```

一个典型的例子


```
  void setPoint(float,float);              // 设置坐标值
  float getX( ) const {return x;}          // 读 x 坐标
  float getY( ) const {return y;}          // 读 y 坐标
  friend ostream & operator<<(ostream &,const Point &);// 重载运算符“ <<”
protected:                                 // 受保护成员
  float x,y;
};
// 下面定义 Point 类的成员函数

//Point 的构造函数
Point::Point(float a,float b)              // 对 x,y 初始化
{x=a;y=b;}
// 设置 x 和 y 的坐标值
void Point::setPoint(float a,float b)      // 为 x,y 赋新值
{x=a;y=b;}
// 重载运算符“ <<”，使之能输出点的坐标
ostream & operator<<(ostream &output,const Point &p)
{output<<"["<<p.x<<","<<p.y<<"]"<<endl;
 return output;
}
```

以上完成了基类 Point 类的声明。

一个典型的例子

现在要对上面写的基类声明进行调试，检查它是否有错，为此要写出 main 函数。实际上它是一个测试程序。

```
int main( )
{Point p(3.5,6.4);// 建立 Point 类对象 p
 cout<<″x=″<<p.getX( )<<″,y=″<<p.getY( )<<endl;// 输出 p 的坐标值
 p.setPoint(8.5,6.8);                    // 重新设置 p 的坐标值
 cout<<″p(new):″<<p<<endl;                    // 用重载运算符“ <<” 输出 p 点坐标
}
```

程序编译通过，运行结果为

```
x=3.5,y=6.4
p(new):[8.5,6.8]
```

测试程序检查了基类中各函数的功能，以及运算符重载的作用，证明程序是正确的。

一个典型的例子

(2) 声明派生类 Circle

在上面的基础上，再写出声明派生类 Circle 的部分：

```
class Circle:public Point//circle 是 Point 类的公用派生类
{public:
  Circle(float x=0,float y=0,float r=0); // 构造函数
  void setRadius(float);              // 设置半径值
  float getRadius( ) const;           // 读取半径值
  float area ( ) const;               // 计算圆面积
  friend ostream &operator<<(ostream &,const Circle &);// 重载运算符“ <<”
 private:
  float radius;
};

// 定义构造函数，对圆心坐标和半径初始化
Circle::Circle(float a,float b,float r):Point(a,b),radius(r){ }
// 设置半径值
void Circle::setRadius(float r)
{radius=r;}
// 读取半径值
float Circle::getRadius( ) const {return radius;}
// 计算圆面积
float Circle::area( ) const
```

一个典型的例子


```
{return 3.14159*radius*radius;}
// 重载运算符“ <<”，使之按规定的形式输出圆的信息
ostream &operator<<(ostream &output,const Circle &c)
{output<<"Center=["<<c.x<<","<<c.y<<"],r="<<c.radius<<",area="<<c.area( )<<endl;
 return output;
}
```

为了测试以上 Circle 类的定义，可以写出下面的主函数：

```
int main( )
{Circle c(3.5,6.4,5.2);// 建立 Circle 类对象 c，并给定圆心坐标和半径
 cout<<"original circle:\\nx="<<c.getX()<<", y="<<c.getY()<<", r="<<c.getRadius( )
    <<", area="<<c.area( )<<endl;    // 输出圆心坐标、半径和面积
 c.setRadius(7.5);                  // 设置半径值
 c.setPoint(5,5);                   // 设置圆心坐标值 x,y
 cout<<"new circle:\\n"<<c;         // 用重载运算符“ <<” 输出圆对象的信息
 Point &pRef=c;                     //pRef 是 Point 类的引用变量，被 c 初始化
 cout<<"pRef:"<<pRef;               // 输出 pRef 的信息
 return 0;
} 程序编译通过，运行结果为 original circle:( 输出原来的圆的数据 )
x=3.5, y=6.4, r=5.2, area=84.9486
new circle:                         ( 输出修改后的圆的数据 )
Center=[5,5], r=7.5, area=176.714
pRef:[5,5]                          ( 输出圆的圆心“点”的数据 )
```

一个典型的例子

(3) 声明 Circle 的派生类 Cylinder

前面已从基类 Point 派生出 Circle 类，现在再从 Circle 派生出 Cylinder 类。

```
class Cylinder:public Circle// Cylinder 是 Circle 的公用派生类
{public:
  Cylinder (float x=0,float y=0,float r=0,float h=0);// 构造函数
  void setHeight(float);                  // 设置圆柱高
  float getHeight( ) const;                 // 读取圆柱高
  float area( ) const;                    // 计算圆表面积
  float volume( ) const;                    // 计算圆柱体积
  friend ostream& operator<<(ostream&,const Cylinder&);// 重载运算符“ <<”
 protected:
  float height;                         // 圆柱高
};
// 定义构造函数
Cylinder::Cylinder(float a,float b,float r,float h)
    :Circle(a,b,r),height(h){}
// 设置圆柱高
void Cylinder::setHeight(float h){height=h;}
// 读取圆柱高
float Cylinder::getHeight( ) const {return height;}
```

一个典型的例子


```
// 计算圆表面积
float Cylinder::area( ) const
{ return 2*Circle::area( )+2*3.14159*radius*height;}
// 计算圆柱体积
float Cylinder::volume() const
{return Circle::area()*height;}
// 重载运算符“ <<”
ostream &operator<<(ostream &output,const Cylinder& cy)
{output<<″Center=[″<<cy.x<<″,″<<cy.y<<″],r=″<<cy.radius<<″,h=″<<cy.height
      <<″\\narea=″<<cy.area( )<<″, volume=″<<cy.volume( )<<endl;
 return output;
}
```

可以写出下面的主函数：

```
int main( )
{Cylinder cy1(3.5,6.4,5.2,10);// 定义 Cylinder 类对象 cy1
 cout<<″\\noriginal cylinder:\\nx=″<<cy1.getX( )<<″, y=″<<cy1.getY( )<<″, r=″
    <<cy1.getRadius( )<<″, h=″<<cy1.getHeight( )<<″\\narea=″<<cy1.area()
    <<″,volume=″<<cy1.volume()<<endl;// 用系统定义的运算符“ <<” 输出 cy1 的数据
 cy1.setHeight(15);                  // 设置圆柱高
 cy1.setRadius(7.5);                 // 设置圆半径
 cy1.setPoint(5,5);                 // 设置圆心坐标值 x,y
 cout<<″\\nnew cylinder:\\n″<<cy1;       // 用重载运算符“ <<” 输出 cy1 的数据
 Point &pRef=cy1;                    //pRef 是 Point 类对象的引用变量
```

一个典型的例子


```
 cout<<"\\npRef as a Point:"<<pRef;     //pRef 作为一个“点”输出
 Circle &cRef=cy1;                    //cRef 是 Circle 类对象的引用变量
 cout<<"\\ncRef as a Circle:"<<cRef;    //cRef 作为一个“圆”输出
 return 0;
}
```

运行结果如下：

```
original cylinder:                    ( 输出 cy1 的初始值 )
x=3.5, y=6.4, r=5.2, h=10             ( 圆心坐标 x,y 。半径 r ， 高 h)
area=496.623, volume=849.486          ( 圆柱表面积 area 和体积 volume)

new cylinder:                         ( 输出 cy1 的新值 )
Center=[5,5], r=7.5, h=15             ( 以 [5,5] 形式输出圆心坐标 )
area=1060.29, volume=2650.72          ( 圆柱表面积 area 和体积 volume)

pRef as a Point:[5,5]                 (pRef 作为一个“点”输出 )
cRef as a Circle: Center=[5,5], r=7.5, area=176.714(cRef 作为一个“圆”输出 )
```

在本例中存在静态多态性，这是运算符重载引起的。可以看到，在编译时编译系统即可以判定应调用哪个重载运算符函数。稍后将在此基础上讨论动态多态性问题。

6.3 不同类型数据间的转换

1. 标准类型数据间的转换

类型不一致时，低的转换为高的；

强制转换法：（类型名）表达式；

2. 用转换构造函数实现类型转换

```
Complex(double r) {real=r;imag=0;}
```

3. 类型转换函数

```
operator double() {return real;}
```

6.3 不同类型数据间的转换

- 转换运算符函数用于将一个类对象转换成内置数据类型或其他类对象。转换运算符成员函数与它要转换成的数据类型具有相同的名字。
- 转换运算符函数与其他重载运算符函数有两点不同：
  - 第一，转换运算符函数无需实参；
  - 第二，转换运算符函数没有返回类型，甚至 void 也不行。可以根据转换运算符函数的名字推出函数的返回类型，例如，如果要将一个 time24 对象转换为 int 类型，则在类中定义该转换运算符函数的名字为 operator int 。

// **Program Example 【例 6-9 】**

```
// Demonstration of a class conversion operator.
...
class time24 // A simple 24 hour time class.
{
public:
        time24( int h = 0, int m = 0, int s = 0 ) ;
        void set_time( int h, int m, int s ) ;
        void get_time( int& h, int& m, int& s ) const ;
        time24 operator+( int secs ) const ;
        time24 operator+( const time24& t ) const ;
        time24& operator++() ; // prefix.
        time24 operator++( int ) ; // postfix.
        bool operator == ( const time24& t ) const ;
        operator int() ;
private:
        int hours ; // 0 to 23
        int minutes ; // 0 to 59
        int seconds ; // 0 to 59
} ;
```

```
time24::operator int()
{
        int no_of_seconds = hours * 3600 + minutes * 60 + seconds ;
       return no_of_seconds ;
}

main()
{
        time24 t( 1, 2, 3 ) ;
        int s ;

        s = t ; // Conversion from a time24 data type to an int data type.
        cout << "Time = " << t
             << "Equivalent number of seconds = " << s << endl ;
}
```

```
Time = 01:02:03
Equivalent number of seconds = 3723
```

6.4

虚函数

- 静态联编举例【例 6-10 】
  - 声明一个基类 Shape ， 公有派生出 Rectangle 和 Circle ， 再由 Rectangle 派生出 Square 。每个类都有同名的 getArea() 函数计算对象的面积。

```
#include <iostream>
using namespace std;
class Shape  // 形状类
{
  public:
    Shape(){}
    ~Shape(){}
    float getArea()const {return 0;}
};
```

引例

```
class Circle:public Shape
{
  public:
    Circle(float Radius):radius(Radius){}
    Circle(){}
    float getArea()const{return 3.14*radius*radius;}
    float getRadius() const {return radius;}
    void setRadius(float Radius) { radius = Radius; }
  private:
    float radius;
};
```

引例

```
class Rectangle:public Shape
{
  public:
    Rectangle(float Length,float
      Width):length(Length),width(Width){ };
    ~Rectangle(){}
    float getArea()const {return length*width;}
    float getLength()const { return length; }
    float getWidth()const { return width; }
    void setLength(float Length) { length = Length; }
    void setWidth(float Width) { width = Width; }
  private:
    float length;
    float width;
};
```

引例

```
class Square: public Rectangle // 正方形类    {
  public:
    Square(float Side): Rectangle(Side,Side) {}
    ~Square() {}
    void setLength(float Length)
      { Rectangle::setLength(Length);Rectangle::setWidth(Length); }
    void setWidth(float Width)
      { Rectangle::setLength(Width);Rectangle::setWidth(Width); }
    void setSide(float Side) // 设置边长。
    {
       setLength(Side);  setWidth(Side);
     }
    float getSide() const // 获取边长。
    { return getWidth();}
    float getArea()const {return Rectangle::getArea();}
};
```

引例


```
int main()    {
  Shape *sp;
  Circle c1(5);
  cout<<" 圆的面积是 "<<c1.getArea()<<endl;
  sp=&c1;
  cout <<" 通过 Shape 指针访问，圆的面积是 " <<          sp->getArea()<<endl;
  Rectangle r1(4,6);
  cout<<" 长方形的面积是 "<<r1.getArea()<<endl;
  sp=&r1;
  cout<<" 通过 Shape 指针访问，长方形的面积是 "<<        sp->getArea()<<endl;
  Square s1(5);
  cout<<" 正方形的面积是 "<<s1.getArea()<<endl;
  sp=&s1;
  cout<<" 通过 Shape 指针访问，正方形的面积是 "<<sp->getArea()<<endl;
  return 0;    }
```

引例

- 运行结果：

圆的面积是 78.5
通过 Shape 指针访问，圆的面积是 0
长方形的面积是 24
通过 Shape 指针访问，长方形的面积是 0
正方形的面积是 25
通过 Shape 指针访问，正方形的面积是 0
Press any key to continue

- 如何按 sp 实际指向的对象，对 sp->getArea() 采用动态联编?

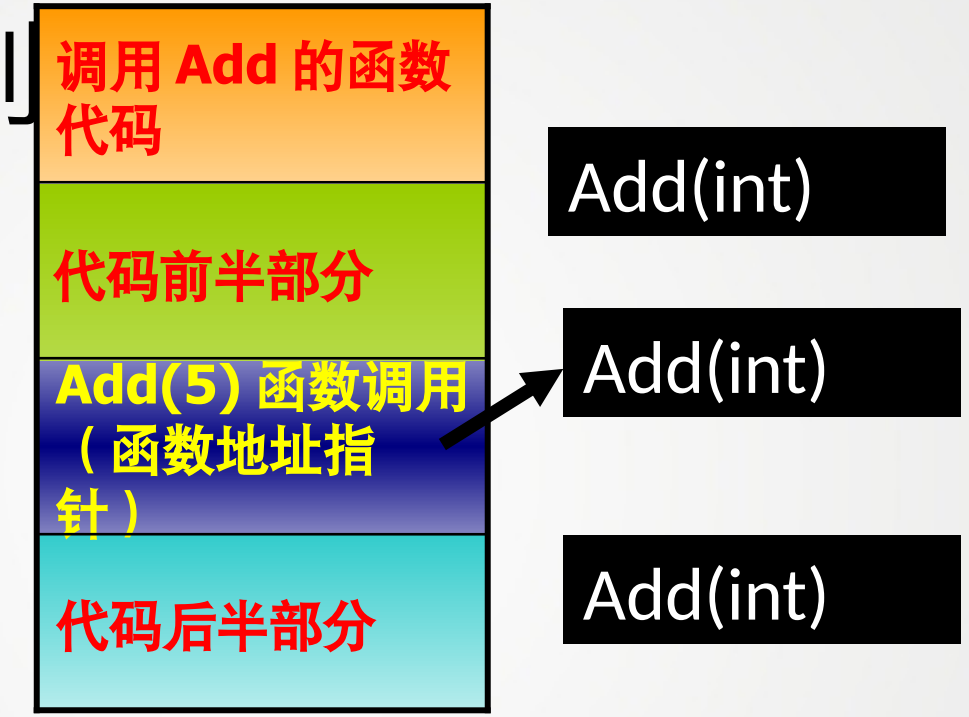
通过在基类中将同名的成员函数设为虚函数，当用基类的指引或引用指向派生类时，实际运行时调用的是实际指向或引用的对象的相应函数而不是基类的同名函数。

6.4 虚函数

- 概念：虚函数是被 virtual 关键字修饰的成员函数
- 格式： virtual 函数类型 函数名 (参数表);
- 虚函数是重载的另一种表现形式（动态重载——更灵活的多态机制）
- 实现机制：通过函数指针，在运行时建立函数调用和函数体间的联系，然后再执行相应的动作（编译器特殊处理）

- 虚函数的机制

```
virtual void Add(int)
{ //..... }
virtual void Add(int)
{ //..... }
virtual void Add(int)
{ //..... }
```



函数的调用通过指针来完成，可以在运行时根据需要改变其执行的代码！

6.4.1 虚函数的定义

虚函数的定义是在基类中进行的，它是在某基类中声明为 virtual 并在一个或多个派生类中被重新定义的成员函数。

虚函数是一个成员函数，在基类的类定义中定义虚函数的一般形式：

```
class 基类名 {
        .....
        virtual 返回值类型 将要在派生类中重载的函数名（参数列表）；
};
```

例如，将类 Student 中的 display() 成员函数定义为虚函数：

```
class Student {
    virtual void display(); // 定义虚函数
    }
```

当基类中的某个成员函数被声明为虚函数后，它就可以在基类的派生类中对虚函数重新定义。在派生类中重新定义的函数应与虚函数具有相同的形参个数和形参类型。以实现统一的接口，不同的定义过程。如果在派生类中没有对虚函数重新定义，则它继承其基类的虚函数。当程序发现虚函数名前的关键字 virtual 后，会自动将其作为动态联编处理，即在程序运行时动态地选择合适的成员函数。

6.4.1 虚函数的定义

- 定义：在类中用 virtual 关键字修饰，可以在派生类中重新定义（多态）
- 注意：在派生类中重新定义（重载）时，其函数原型，包括返回类型、函数名、参数个数与参数类型及其顺序，都必须与基类中的原型相同。
- 原因：虚函数用函数指针实现

6.4.1 虚函数的定义

虚函数的作用是允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问派生类中的同名函数。

【例 6-11 】基类与派生类中有同名函数

```
// 声明基类 Student
class Student
{public:
 Student(int,string,float);
 void display();
  protected:
  int num;
  string name;
  float score;
};
```

【例 6-11】基类与派生类中有同名函数（续）

```
//Student 类成员函数的实现
Student::Student(int n,string nam,float s)
{num =n; name=nam; score = s;}

void Student ::display()
{ cout<<"num:"<<num<<"\name:"<<name<<"\nscore:"<<score<<"\n\n";}
 // 声明公有派生类 Graduate
class Graduate:public Student
{public:
    Graduate(int,string,float,float);
    void display();
private:
    float pay;
};
```

【例 6-11 】基类与派生类中有同名函数 (续)

```
//Graduate 类成员函数的实现
void Graduate::display()
{ cout<<"num:"<<num<<"\nname:"<<name<<"\nscore:"<<score<<"\
      npay="<<pay<<endl;}
 Graduate::Graduate(int n, string nam,float s,float p):Student(n,nam,s),pay(p){ }
 // 主函数
int main()
{ Student stud1(1001,"Li",87.5);
 Graduate grad1(2001,"Wang",98.5,563.5);
 Student *pt=&stud1;
 pt->display();
 pt=&grad1;
 pt->display();
 getchar();
 return 0;
}
```

运行结果如下，请仔细分析。

```
num:1001(stud1 的数据 )
name:Li
score:87.5

num:2001          (grad1 中基类部分的数据 )
name:wang
score:98.5
```

下面对程序作一点修改，在 Student 类中声明 display 函数时，在最左面加一个关键字 virtual ，即

```
virtual void display( );
```

这样就把 Student 类的 display 函数声明为虚函数。程序其他部分都不改动。再编译和运行程序，请注意分析运行结果 :

```
num:1001(stud1 的数据 )
name:Li
score:87.5
num:2001          (grad1 中基类部分的数据 )
name:wang
score:98.5
pay=1200          ( 这一项以前是没有的 )
```

定义虚函数时要遵循下列规则

（1）只有成员函数才能声明为虚函数，因为虚函数仅适用于有继承关系的类对象，所以普通函数和友元函数都不能声明为虚函数。

（2）虚函数的声明只能出现在类声明中的函数原型声明或定义中，在类外定义时不能出现 virtual 关键字。

（3）通过定义虚函数来使用 C++ 语言提供的多态性机制时，派生类应该是从基类公有派生的。

（4）类的静态成员函数不可以声明为虚函数，因为静态成员函数不受限于某个对象。

（5）类的构造函数不可以是虚函数。虚函数是为了实现动态多态性，根据不同的对象在运行过程中才能决定和哪个函数建立关联，而构造函数是在对象创建时运行的，故虚构造函数是没有意义的。

（6）析构函数可以声明为虚函数，而且通常被声明为虚函数。

（7）内联函数不能声明为虚函数，因为内联函数不能在运行中动态确定其位置。

（8）基类的虚函数无论被公有继承多少次，在多级派生类中仍然为虚函数。

虚函数与重载函数的关系

一般的函数重载时，只要函数名相同，函数的参数个数、参数类型或顺序必须不同，函数的返回类型也可以不同。但是，当重载一个虚函数时，也就是说在派生类中重新定义此虚函数时，要求函数名、返回类型、参数个数、参数类型以及参数的顺序都必须与基类中的虚函数原型完全相同。如果仅仅是返回类型不同，其余均相同，系统会给出错误信息；若仅仅是函数名相同，而参数的个数、类型或顺序不同，则系统将它作为普通的函数重载，这时将丢失虚函数的特性。

【例 6-12 】虚函数与重载函数的比较。

```
#include <iostream>
using namespace std;
class Base
{
public:
```

虚函数与重载函数的关系


```
    virtual void fun1(){cout<<"--Base fun1--\n";}
    virtual void fun2(){cout<<"--Base fun2--\n";}
    virtual void fun3(){cout<<"--Base fun3--\n";}
    void fun4(){cout<<"--Base fun4--\n";}
};
class Derived:public Base
{
public:
    virtual void fun1(){cout<<"--Derived fun1--\n";} //fun1() 是虚函数
    void fun2(int x){cout<<"--Derived fun2--\n";}   //fun2() 作为一般函数重载，          虚特性消失
  char fun3(){cout<<"--Derived fun3--\n";}  // 编译错误，因为只有返回类型不同
    void fun4(){cout<<"--Derived fun4--\n";} //fun4() 是一般函数重载，不是虚函数
};
```

虚函数与重载函数的关系


```
int main()
{
        Base obj1,*p1;
        Derived obj2;
        p1=&obj2;
        p1->fun1();                    // 调用 Derived::fun1()
        p1->fun2();                    // 调用 Base::fun2()
        p1->fun4();                    // 调用 Base::fun4()
        return 0;
}
```

6.4.2 虚函数的作用

虚函数的作用是实现动态联编，也就是在程序的运行阶段动态地选择合适的成员函数。实现动态关联需要 3 个条件：

（1）必须把需要动态关联的行为定义为类的公共属性的虚函数；

（2）类之间存在子类型关系，一般表现为一个类从另一个类公有派生而来；

（3）必须先使用基类指针指向子类型的对象，然后直接或者间接使用基类指针调用虚函数。

因此，实现动态关联，只能通过指向基类的指针或基类对象的引用来调用虚函数，其格式如下：

（1）指向基类的指针变量名 -> 虚函数名（实参表）

（2）基类对象的引用名 . 虚函数名（实参表）

下面通过比较例 6-7 和例 6-8 两个实例的运行结果理解虚函数的作用。其中，例 6-7 中没有将基类与派生类中的同名函数设为虚函数，例 6-8 中将基类与派生中的同函数设为虚函数。

- 动态联编举例
  - 仅需将引例中的顶层基类 Shape 的 getArea() 函数加上 virtual 声明

```
class Shape  // 形状类
{
  public:
    Shape(){}
    ~Shape(){}
    virtual float getArea()const {return 0;}
};
```

6.4.2 虚函数的作用

```
int main()
{
  Circle c1(5);
  cout<<" 圆的面积是 "<<c1.getArea()<<endl;

  Shape *pShape=&c1;
  cout<<" 通过 Shape 指针访问，圆的面积是 "<<        pShape->getArea()<<endl;

  Shape &rShape=c1;
  cout<<" 通过 Shape 引用访问，圆的面积是 "<<rShape.getArea()<<endl;

  Shape oShape=c1;
  cout<<" 通过 Shape 对象访问，圆的面积是 "<<oShape.getArea()<<endl;
  return 0;
}
```

6.4.2 虚函数的作用

- 运行结果：

圆的面积是 78.5
通过 Shape 指针访问，圆的面积是 78.5
通过 Shape 引用访问，圆的面积是 78.5
通过 Shape 对象访问，圆的面积是 0
Press any key to continue

- 使用虚函数是实现动态联编的基础。正确使用虚函数，需要满足三个条件：

（1）具有符合类型兼容规则的公有派生类层次结构。
（2）在派生类中重新定义基类中的虚函数，对其进行覆盖（override）。
（3）通过基类指针或基类引用访问虚函数。

6.4.2 虚函数的作用

【例 6-13】 将基类 Student 与派生类 Graduate 中的同名函数 display() 设为虚函数。

```
// 声明基类 Student
class Student
{public:
    Student(int,string,float);
    virtual void display();
 protected:
    int num;
    string name;
    float score;
};
//Student 类成员函数的实现
Student::Student(int n,string nam,float s)
{num =n; name=nam; score = s;}

void Student ::display()
{ cout<<"num:"<<num<<"\name:"<<name<<"\nscore:"<<score<<"\n\n";}
```

6.4.2 虚函数的作用

【例 6-13】 将基类 Student 与派生类 Graduate 中的同名函数 display() 设为虚函数。(续)

```
// 声明公有派生类 Graduate
class Graduate:public Student
{public:
    Graduate(int,string,float,float);
    virtual void display();
private:
    float pay;
};
//Graduate 类成员函数的实现
void Graduate::display()
{ cout<<"num:"<<num<<"\nname:"<<name<<"\nscore:"<<score<<"\
      npay="<<pay<<endl;}
Graduate::Graduate(int n, string nam,float s,float
      p):Student(n,nam,s),pay(p){ }
```

6.4.2 虚函数的作用

【例 6-13】 将基类 Student 与派生类 Graduate 中的同名函数 display() 设为虚函数。(续)

```
// 主函数
int main()
{
    Student stud1(1001,"Li",87.5);
    Graduate grad1(2001,"Wang",98.5,563.5);
    Student *pt=&stud1;
    pt->display();
    pt=&grad1;
    pt->display();
    getchar();
    return 0;
}
```

6.4.2 虚函数的作用

- 观察【例 6-13 】改动后程序的运行结果可以发现，在输出 grad1 的信息时输出了其所得助学金额的金额，即用同一种调用形式 pt->display() 当指向学生 stud1 时输出了 stud1 的全部数据，当指向 grad1 时输出了研究生 grad1 的全部数据。 pt 一个基类指针，可以调用同一类族中不同类的虚函数，这就是多态性，对同一消息，不同对象有不同的响应方式。虚函数的奇妙作用在于实现动态多态。

6.4.2 虚函数的作用

- 基类的指针是用来指向基类对象的，如果用它指向派生类对象，则进行指针类型转换，将派生类对象的指针先转换为基类的指针，所以基类的指针指向的是派生类对象中的基类部分。如果不设置虚函数，是无法通过基类指针去调用派生类对象中的成员函数。虚函数突破了这一限制，在派生类的基类部分，派生类的函数取代了基类原来的函数，因此在使用基类指针指向派生类对象后，调用基类函数时就调用了派生类的同名函数。

6.4.2 虚函数的作用

- 虚函数的以上功能是很有实用意义的。在面向对象的程序设计中，经常会用到类的继承，保留基类的特性，以减少新类开发的时间。但是，从基类继承来的某些成员函数不完全适应派生类的需要。例如在例 6-7 中，基类的 display 函数只输出基类的数据，而派生类的 display 函数需要输出派生类的数据。过去我们曾经使派生类的输出函数与基类的输出函数不同名（如 display 和 display1 ），如果派生的层次多，就要起许多不同的函数名。利用虚函数就很好地解决了这个问题。可以看到：当把基类的某个成员函数声明为虚函数后，允许在其派生类中对该函数的定义进行覆盖，赋予它新的功能，并且在通过指向基类的指针指向同一类族中不同类的对象时调用所指向类的同名函数。

6.4.2 虚函数的作用

- 由虚函数实现的动态多态就是同一类族中不同类的对象，对同一函数调用作出不同的响应。那么，在什么情况下把一个成员函数声明为虚函数呢？主要考虑以下几点。
- （1）首先看成员函数所在的类是否会作为基类，然后看成员函数在类的继承后是否要更改功能，如果希望更改其功能，一般应该将它声明为虚函数。
- （2）如果成员函数在类被继承后功能不需修改，或派生类用不到该函数，则不要声明为虚函数。不要把基类中的所有的成员函数都声明为虚函数。
- （3）应考虑对成员函数的调用是通过对象名还是通过基类的指针或引用去访问，如果是通过基类的指针或引用去访问，则应当声明为虚函数。仅仅是通过对象名去访问派生类时，没有必要声明为虚函数。
- （4）有时，在定义虚函数时，并不定义其函数体，即函数体是空的。它的作用只是定义了一个虚函数名，具体功能留给派生类去添加。这时需要

在使用虚函数时，有两点要注意：

(1) 只能用 virtual 声明类的成员函数，使它成为虚函数，而不能将类外的普通函数声明为虚函数。因为虚函数的作用是允许在派生类中对基类的虚函数重新定义。显然，它只能用于类的继承层次结构中。

(2) 一个成员函数被声明为虚函数后，在同一类族中的类就不能再定义一个非 virtual 的但与该虚函数具有相同的参数和函数返回值类型的同名函数。

根据什么考虑是否把一个成员函数声明为虚函数呢？主要考虑以下几点：

(1) 首先看成员函数所在的类是否会作为基类，然后看成员函数在类的继承后有是否要更改功能，如果希望更改其功能，一般应该将它声明为虚函数。

(2) 如果成员函数在类被继承后功能不需修改，或派生类用不到该函数，则不要声明为虚函数。不要把基类中的所有的成员函数都声明为虚函数。

(3) 应考虑对成员函数的调用是通过对象名还是通过基类的指针或引用去访问，如果是通过基类的指针或引用去访问，则应当声明为虚函数。仅仅是通过对象名去访问派生类时，没有必要声明为虚函数。

(4) 有时，在定义虚函数时，并不定义其函数体，即函数体是空的。它的作用只是定义了一个虚函数名，具体功能留给派生类去添加。这时需要将虚函数设为纯虚函数，包含纯虚函数的类称为抽象类，在 6.5 节对此进行详细讨

6.4.3 对象的存储

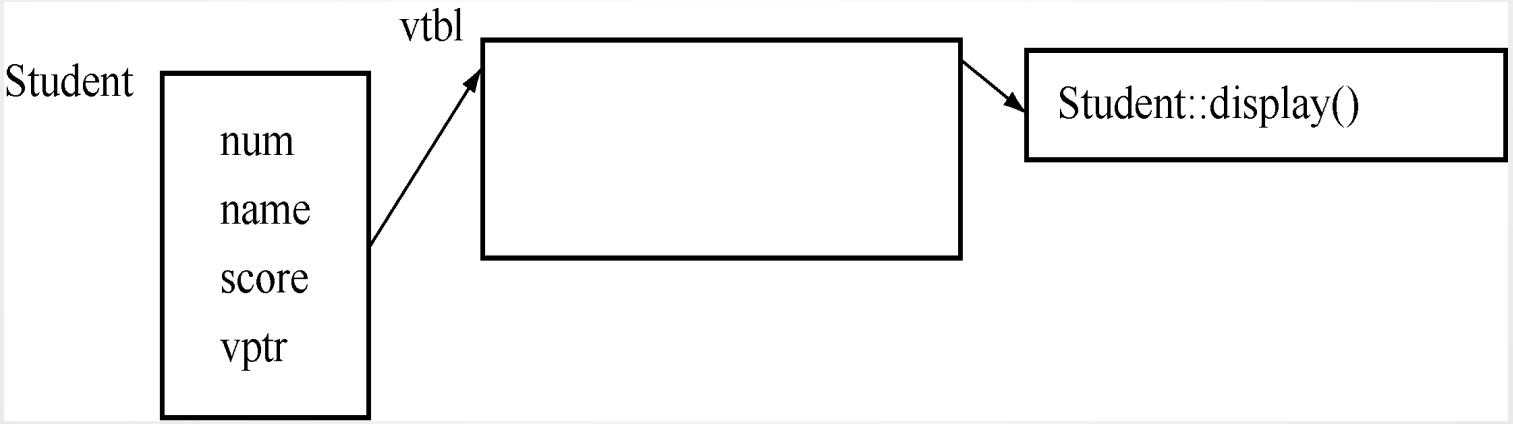
对象是如何在内存中存储的？当定义一个对象时，它的数据成员按顺序存储在内存中。当有派生类对象时，新增的数据成员加在基类的数据成员之后。

为了管理虚函数的调用，需要在对象中增加一个数据项，用来说明在调用虚函数时具体调用的是哪个函数。通常这一数据项是 vtbl （ virtual table ， 虚函数表）的地址，称为 vptr （ virtual pointer ， 虚函数指针）。

6.4.3 对象的存储

下面以一个例子分析虚函数的实现。 Graduate 是 Student 的派生类，Graduate 可以看成是一种特殊的 Student ， 可以被当成 Student 。另外， Graduate 还具有它自己的数据成员。

为了控制虚函数的调用，我们在 Student 类中需要设置一个 vtbl 表来告知 Student 对象在调用 display 函数时是哪个函数被调用，如图 6-1 所示。



虚函数的使用只是增加了两次访存，并不会过多影响程序的执行速度；在存储上每个类多了一个 vtbl 表，并没有过多增加内存。通过上面的分析我们知道执行到底慢多少，需要的存储到底大多少，满足一些读者的好奇及消除人们在设计时的恐惧心理。

6.4.4 虚析构函数

以前曾经介绍过，析构函数的作用是在对象撤销之前做必要的“清理现场”的工作。当派生类的对象从内存中撤销时一般先运行派生类的析构函数，然后再调用基类的析构函数。如果用 new 运算符建立了派生类的临时对象，对指向基类的指针指向这个临时对象，当用 delete 运算符撤销对象时，系统执行的是基类的析构函数，而不是派生类的析构函数，不能彻底完成“清理现场”的工作。解决的办法是将基类及派生类的析构函数设为虚函数，这时无论基类指针指的是同一类族中的哪一个类对象，系统会采用动态关联，调用相应的析构函数，对该对象进行清理工作，符合人们的愿望。

- 虚析构函数
  - 构造函数不能声明为虚函数
    - 虚函数主要是针对对象的，而构造函数是在对象产生之前运行的，故虚构造函数没意义
  - 析构函数通常有必要声明为虚函数
    - 原型声明： virtual ~ 类名 ();
    - 当可能通过基类指针删除派生类对象时
    - 在对指向动态分配对象 (由 new 操作产生) 的指针进行 delete 操作时，隐含着对析构函数的调用，而该指针的声明类型可能是对象类型的基类

虚析构函数举例

```
#include <iostream>
using namespace std;
class A
{
  public:
    A()
    {
      cout<<"A 类构造函数调用完毕 "<<endl;
    }
    virtual ~A()
    {
      cout<<"A 类析构函数调用完毕 "<<endl;
    }
};
```

虚析构函数举例

```
class B : public A  {
  private:
     char *str;
  public:
     B(const char *const Str)
     {
          if (Str) {      str = new char[strlen(Str) +1];  strcpy(str,Str);     }
          else {      str = new char[1];  str[0] = '\0';      }
          cout<<"B 类构造函数调用完毕，空间已申请 "<<endl;
     }
     virtual~B()  {
               delete[ ] str;
               cout<<"B 类析构函数调用完毕，空间已释放 "<<endl;
      }
};
```

虚析构函数举例

```
int main()
{
     A *pa=new B("Hello!");
    delete pa;
    return 0;
}
```

- 运行结果：

A 类构造函数调用完毕
B 类构造函数调用完毕，空间已申请
B 类析构函数调用完毕，空间已释放
A 类析构函数调用完毕
Press any key to continue

6.5 纯虚函数与抽象类

6.5.1 纯虚函数

纯虚函数的一般定义形式为：

```
class 类名
  {
      virtual 返回值类型 函数名（参数表） = 0;
      ……
    };
```

可见，将一个虚函数声明为纯虚函数，需要在虚函数原型的语句结束符“；”之前加上 =0 ，没有代码。例如，设计一个 Shape 基类，并在此基础上派生出 Circle 类。

```
class Shape{
    public: virtual double area()=0 ;   //=0 表示函数是一个纯虚函数;
};
```

这里 Shape 类的 area() 中不仅有 virtual ，还有 =0 ，表示 area() 是一个纯虚函数，包含纯虚函数的类是一个抽象类，不能定义抽象类的对象。此时 Shape 是一个抽象类，不能定义 Shape 类的对象。

6.5.1 纯虚函数

纯虚函数的一般定义形式为：

```
class 类名
  {
      virtual 返回值类型 函数名（参数表） = 0;
      ……
    };
```

可见，将一个虚函数声明为纯虚函数，需要在虚函数原型的语句结束符“；”之前加上 =0 。例如，设计一个 Shape 基类，并在此基础上派生出 Circle 类。

```
class Shape{
    public: virtual double area()=0 ；   //=0 表示函数是一个纯虚函数；
};
```

这里 Shape 类的 area() 中不仅有 virtual ，还有 =0 ，表示 area() 是一个纯虚函数，包含纯虚函数的类是一个抽象类，不能定义抽象类的对象。此时 Shape 是一个抽象类，不能定义 Shape 类的对象。

6.5.1 纯虚函数

【例 6-14 】分析程序执行结果。

```
#include<iostream>
using namespace std;
class Staff                                  // 职员类
{
public:
        virtual void Pay()=0;                // 声明为纯虚函数
};
class CommonWorker:public Staff              // 普通员工类
{
public:
        CommonWorker(double w,double b):wage(w),bonus(b){}
        virtual void Pay()                   // 重新定义虚函数
        {
                cout<<" 基本工资 + 奖金 ="<<wage+bonus<<endl;
        }
```

6.5.1 纯虚函数

```
protected:
        double wage;
        double bonus;
};
class Manager:public CommonWorker  // 经理类
{
public:
        Manager(double w,double b,double a):CommonWorker(w,b),allowance(a){}
        virtual void Pay()              // 重新定义虚函数
        {
                cout<<" 基本工资 + 奖金 + 职务津贴 ="<<wage+bonus+allowance<<endl;
        }
protected:
        double allowance;
};
```

6.5.1 纯虚函数


```
int main()
{
        Staff *s;                       // 基类指针
        CommonWorker c1(800,2000);
        Manager m1(1200,2000,500);
        s=&c1;                          // 基类指针指向派生类对象
        s->Pay();                               // 调用 CommonWorker 类的 Pay() 函数
        s=&m1;                          // 间接基类指针指向派生类对象
        s->Pay();                               // 调用 Manager 类的 Pay() 函数
   return 0;
}
```

6.5.2 抽象类

如果一个类中至少有一个**纯虚函数**，这个类就是为**抽象类**，通常也称为抽象基类。它的主要作用是为一个类族提供统一的公共接口，使它们更有效地发挥多态性的特性。**使用抽象类时需注意以下几点**：

- （1）抽象类只能作为用作其他类的基类，不能建立抽象类的对象。抽象类处于继承层次结构的较上层，一个抽象类自身无法实例化，而只能通过继承机制，生成抽象类的非抽象派生类，然后再实例化。
- （2）抽象类不能用作参数类型、函数返回值或显式转换的类型。
- （3）抽象类不能定义对象，但是可以声明一个抽象类的指针和引用。通过指针或引用可以指向并访问派生类对象，以访问派生类的成员。
- （4）抽象类派生出新的类之后，如果派生类给出所有纯虚函数的函数实现，这个派生类就可以声明自己的对象，因而不再是抽象类；反之，如果派生类没有给出全部纯虚函数的实现，这时的派生类仍然是一个抽象类。

6.5.2 抽象类

虽然抽象类不能定义对象(或者说抽象类不能实例化),但是可以定义指向抽象类对象的指针变量。当派生类成为具体类之后,就可以用这种指针指向派生类对象,然后通过该指针调用虚函数,实现多态的操作。

6.5.2 抽象类

```
// 抽象类举例。
class Shapes
{
public:
    virtual void draw() = 0;                    // 纯虚函数
    virtual void rotate(int) = 0;               // 纯虚函数
};
class circle: public Shapes
{
private:
    double radius;
public:
    circle(int r);
    void draw() { }
    void rotate(int) { }
    double area(){return 3.14159*radius*radius; }
    double volume(){return 3*3.14159*radius*radius*radius/4;}
};
```

虚函数和抽象基类的应用

【例 6-15 】以 Point 为基类的点—圆—圆柱体类的层次结构。现在要对它进行改写，在程序中使用虚函数和抽象基类。类的层次结构的顶层是抽象基类 Shape(形状) 。 Point(点), Circle(圆), Cylinder(圆柱体) 都是 Shape 类的直接派生类和间接派生类。

下面是一个完整的程序，为了便于阅读，分段插入了一些文字说明。

程序如下 :

第 (1) 部分

```
#include <iostream>
using namespace std;
// 声明抽象基类 Shape
class Shape
{public:
 virtual float area( ) const {return 0.0;}// 虚函数
 virtual float volume() const {return 0.0;}     // 虚函数
 virtual void shapeName() const =0;             // 纯虚函数
};
```

第 (2) 部分

```
// 声明 Point 类
class Point:public Shape//Point 是 Shape 的公用派生类
{public:
  Point(float=0,float=0);
  void setPoint(float,float);
  float getX( ) const {return x;}
  float getY( ) const {return y;}
  virtual void shapeName( ) const {cout<<"Point:";}     // 对虚函数进行再定义
  friend ostream & operator<<(ostream &,const Point &);
```

```
};
// 定义 Point 类成员函数
Point::Point(float a,float b)
{x=a;y=b;}

void Point::setPoint(float a,float b)
{x=a;y=b;}

ostream & operator<<(ostream &output,const Point &p)
{output<<"["<<p.x<<","<<p.y<<"]";
return output;
}
```

第 (3) 部分

```
// 声明 Circle 类
class Circle:public Point
{public:
  Circle(float x=0,float y=0,float r=0);
  void setRadius(float);
  float getRadius( ) const;
  virtual float area( ) const;
  virtual void shapeName( ) const {cout<<"Circle:";}// 对虚函数进行再定义
  friend ostream &operator<<(ostream &,const Circle &);
protected:
```

```
  float radius;
};
// 声明 Circle 类成员函数
Circle::Circle(float a,float b,float r):Point(a,b),radius(r){ }

void Circle::setRadius(float r):radius(r){ }

float Circle::getRadius( ) const {return radius;}

float Circle::area( ) const {return 3.14159*radius*radius;}

ostream &operator<<(ostream &output,const Circle &c)
{output<<"["<<c.x<<","<<c.y<<"], r="<<c.radius;
 return output;
}
```

第 (4) 部分

```
// 声明 Cylinder 类
class Cylinder:public Circle
{public:
  Cylinder (float x=0,float y=0,float r=0,float h=0);
  void setHeight(float);
  virtual float area( ) const;
  virtual float volume( ) const;
```

```
  virtual void shapeName( ) const {cout<<"Cylinder:";}// 对虚函数进行再定义
  friend ostream& operator<<(ostream&,const Cylinder&);
 protected:
  float height;
};
// 定义 Cylinder 类成员函数
Cylinder::Cylinder(float a,float b,float r,float h)
    :Circle(a,b,r),height(h){ }

void Cylinder::setHeight(float h){height=h;}

float Cylinder::area( ) const
{ return 2*Circle::area( )+2*3.14159*radius*height;}

float Cylinder::volume( ) const
{return Circle::area( )*height;}

ostream &operator<<(ostream &output,const Cylinder& cy)
{output<<"["<<cy.x<<","<<cy.y<<"], r="<<cy.radius<<", h="<<cy.height;
return output;
}
```

第 (5) 部分

```
//main 函数
int main( )
{Point point(3.2,4.5);// 建立 Point 类对象 point
 Circle circle(2.4,1.2,5.6);                  // 建立 Circle 类对象 circle
 Cylinder cylinder(3.5,6.4,5.2,10.5);         // 建立 Cylinder 类对象 cylinder
 point.shapeName();                           // 静态关联
 cout<<point<<endl;

 circle.shapeName();                          // 静态关联
 cout<<circle<<endl;

 cylinder.shapeName();                        // 静态关联
 cout<<cylinder<<endl<<endl;

 Shape *pt;                                   // 定义基类指针
```

虚函数和抽象基类的应用

```
pt=&point;                          // 指针指向 Point 类对象
pt->shapeName( );                   // 动态关联
cout<<"x="<<point.getX( )<<",y="<<point.getY( )<<"\\narea="<<pt->area( )
   <<"\\nvolume="<<pt->volume()<<"\\n\\n";

pt=&circle;                         // 指针指向 Circle 类对象

pt->shapeName( );                   // 动态关联
cout<<"x="<<circle.getX( )<<",y="<<circle.getY( )<<"\\narea="<<pt->area( )
   <<"\\nvolume="<<pt->volume( )<<"\\n\\n";

pt=&cylinder;                       // 指针指向 Cylinder 类对象
pt->shapeName( );                   // 动态关联
cout<<"x="<<cylinder.getX( )<<",y="<<cylinder.getY( )<<"\\narea="<<pt->area( )
   <<"\\nvolume="<<pt->volume( )<<"\\n\\n";
return 0;
}
```

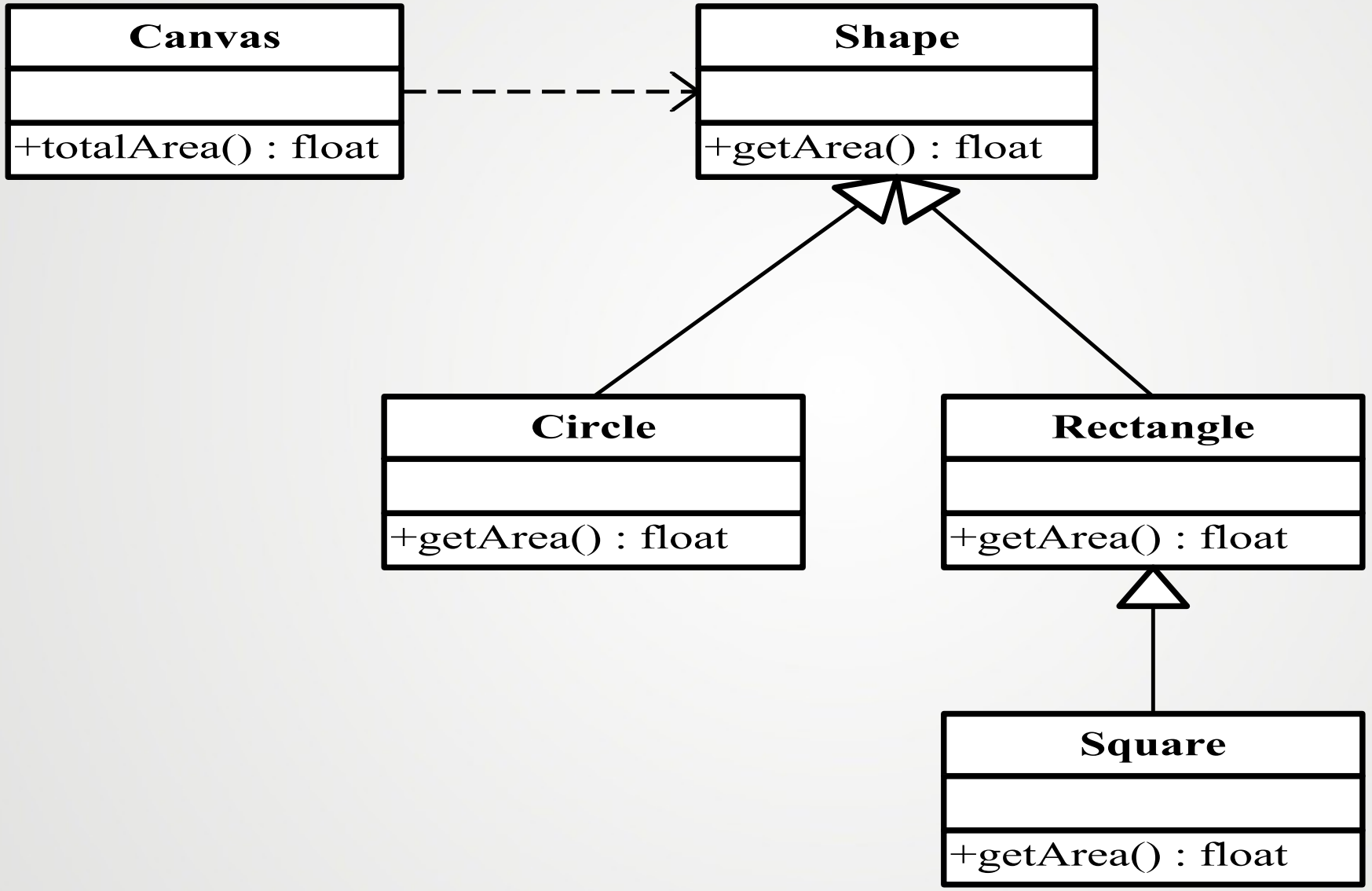
程序运行结果如下

Point:[3.2,4.5](Point 类对象 point 的数据：点的坐标)
Circle:[2.4,1.2], r=5.6　　(Circle 类对象 circle 的数据：圆心和半径)
Cylinder:[3.5,6.4], r=5.5, h=10.5　(Cylinder 类对象 cylinder 的数据：　圆心、半径和高)

Point:x=3.2,y=4.5　　(输出 Point 类对象 point 的数据：点的坐标)
area=0　　(点的面积)
volume=0　　(点的体积)

Circle:x=2.4,y=1.2　　(输出 Circle 类对象 circle 的数据：圆心坐标)
area=98.5203　　(圆的面积)
volume=0　　(圆的体积)
Cylinder:x=3.5,y=6.4　　(输出 Cylinder 类对象 cylinder 的数据：圆心坐标)
area=512.595　　(圆的面积)
volume=891.96　　(圆柱的体积)

- 抽象类举例


Canvas
+totalArea() : float
Shape
+getArea() : float
Circle
+getArea() : float
Rectangle
+getArea() : float
Square
+getArea() : float

• 抽象类举例

```
class Shape  // 形状类
{
public:
      Shape(){}
      ~Shape(){}
      virtual float getArea() const=0;
};
class Circle:public Shape
{
 public:
      Circle(float Radius):radius(Radius){}
      Circle(){}
      float getArea()const{return 3.14*radius*radius;}
      float getRadius() const {return radius;}
  private:
      float radius;
};
```

• 抽象类举例

```
class Rectangle:public Shape
{
  public:
      Rectangle(float Length,float Width):length(Length),width(Width){ };
      ~Rectangle(){}
      float getArea()const {return length*width;}
      float getLength()const { return length; }
      float getWidth()const { return width; }
  private:
      float length;
      float width;
};
```

• 抽象类举例

```
class Square: public Rectangle // 正方形类
{
  public:
      Square(float Side): Rectangle(Side,Side) {}
      ~Square() {}
      float getSide() const // 获取边长。
      {
                return getWidth();
      }
      float getArea()const {return Rectangle::getArea();}
};
```

- 抽象类举例

```
class Canvas
{
 public:
      float totalArea(Shape* s[],int N)const
      {
                float total=0;
                for (int i=0;i<N;i++)
                {
                          total+=s[i]->getArea();
                }
                return total;
      }
};
```

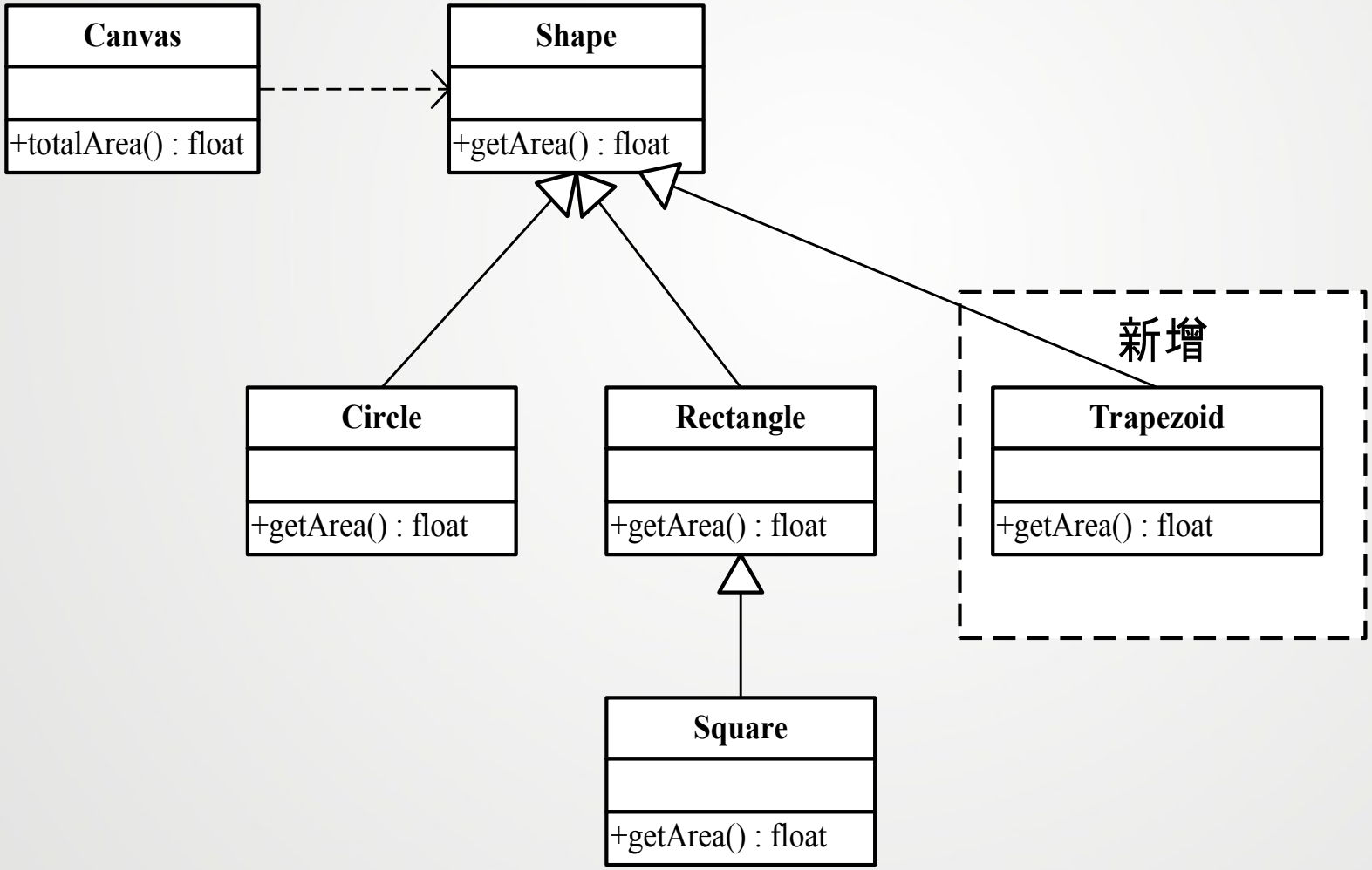
• 抽象类举例

```
int main()   {
      Shape **ppShape=new Shape*[3];
     ppShape[0]=new Circle(5);
     ppShape[1]=new Rectangle(4,6);
     ppShape[2]=new Square(5);
     cout<<" 圆的面积是 "<<ppShape[0]->getArea()<<endl;
     cout<<" 长方形的面积是 "<<ppShape[1]->getArea()<<endl;
     cout<<" 正方形的面积是 "<<ppShape[2]->getArea()<<endl;
     Canvas c1;
     float total=c1.totalArea(ppShape,3);
     cout<<" 总面积是 "<<total<<endl;
     for (int i=0;i<3;i++)
                delete ppShape[i];
     delete[] ppShape;
     return 0;
}
```

运行结果：
圆的面积是 78.5
长方形的面积是 24
正方形的面积是 25
总面积是 127.5
Press any key to continue

抽象类举例

- 在此基础上，增加一个子类 Trapezoid



• 抽象类举例

```
class Trapezoid: public Shape // 梯形类
{
  public:
     Trapezoid(float TopLine,float BottomLine,float
     Height):topLine(TopLine),bottomLine(BottomLine),height(Height){ };
     ~Trapezoid() {}
     float getArea()const {return (topLine+bottomLine)*height/2;}
  private:
     float topLine;
     float bottomLine;
     float height;
};
```

• 抽象类举例

```
int main()  {
     Shape **ppShape=new Shape*[4];
    ppShape[0]=new Circle(5);
    ppShape[1]=new Rectangle(4,6);
    ppShape[2]=new Square(5);
    ppShape[3]=new Trapezoid(7,8,9);
    cout<<" 圆的面积是 "<<ppShape[0]->getArea()<<endl;
    cout<<" 长方形的面积是 "<<ppShape[1]->getArea()<<endl;
    cout<<" 正方形的面积是 "<<ppShape[2]->getArea()<<endl;
    cout<<" 梯形的面积是 "<<ppShape[3]->getArea()<<endl;
    Canvas c1;
    float total=c1.totalArea(ppShape,4);
    cout<<" 总面积是 "<<total<<endl;
    for (int i=0;i<4;i++)
            delete ppShape[i];
    delete[] ppShape;
    return 0;
```

运行结果：

```
圆的面积是 78.5
长方形的面积是 24
正方形的面积是 25
梯形的面积是 67.5
总面积是 195
Press any key to continue
```

• 抽象类举例

定义四个类，它们的继承关系为： Line --> Rec --> Cuboid --> Cube 。

Line 是一个抽象类，也是最顶层的基类，在 Line 类中定义了两个纯虚函数 area() 和 volume() 。

在 Rec 类中，实现了 area() 函数；所谓实现，就是定义了纯虚函数的函数体。但这时 Rec 仍不能被实例化，因为它没有实现继承来的 volume() 函数， volume() 仍然是纯虚函数，所以 Rec 也仍然是抽象类。

直到 Cuboid 类，才实现了 volume() 函数，才是一个完整的类，才可以被实例化。

- 抽象类举例

```
#include <iostream>
using namespace std;
// 线
class Line{
public:
Line(float len);
virtual float area() = 0;
virtual float volume() = 0;
protected:
float m_len;
};
Line::Line(float len): m_len(len){ }
```

• 抽象类举例

```
// 矩形
class Rec: public Line{
public:
Rec(float len, float width);
float area();
protected:
float m_width;
};
Rec::Rec(float len, float width): Line(len), m_width(width){ }
float Rec::area(){ return m_len * m_width; }
```

• 抽象类举例

```
// 长方体
class Cuboid: public Rec{
public:
Cuboid(float len, float width, float height);
float area();
float volume();
protected:
float m_height;
};
Cuboid::Cuboid(float len, float width, float height): Rec(len, width), m_height(height){ }
float Cuboid::area(){ return 2 * ( m_len*m_width + m_len*m_height + m_width*m_height); }
float Cuboid::volume(){ return m_len * m_width * m_height; }
```

- 抽象类举例

```
// 正方体
class Cube: public Cuboid{
public:
Cube(float len);
float area();
float volume();
};
Cube::Cube(float len): Cuboid(len, len, len){ }
float Cube::area(){ return 6 * m_len * m_len; }
float Cube::volume(){ return m_len * m_len * m_len; }
```

- 抽象类举例

```
int main(){
Line *p = new Cuboid(10, 20, 30);
cout<<"The area of Cuboid is "<<p->area()<<endl;
cout<<"The volume of Cuboid is "<<p->volume()<<endl;
p = new Cube(15);
cout<<"The area of Cube is "<<p->area()<<endl;
cout<<"The volume of Cube is "<<p->volume()<<endl;
return 0;
}
```

- 抽象类举例

运行结果：

```
The area of Cuboid is 2200
The volume of Cuboid is 6000
The area of Cube is 1350
The volume of Cube is 3375
```

• 抽象类举例

- 在实际开发中，你可以定义一个抽象基类，只完成部分功能，未完成的功能交给派生类去实现（谁派生谁实现）。这部分未完成的功能，往往是基类不需要的，或者在基类中无法实现的。虽然抽象基类没有完成，但是却强制要求派生类完成，这就是抽象基类的“霸王条款”。

6.5

知识扩展

- 运行时类型识别（RTTI:Run-time type identification）
  - 是指在运行时通过基类指针（或引用）辨别对象所属的具体派生类
  - 只对多态类型适用
  - 比虚函数动态绑定的开销更大，当需求变化超出了设计基类时的预期，又不能修改基类时才使用
  - 两种运行时类型识别机制：
    - dynamic_cast 运算符：做类型转换的尝试
    - typeid 运算符：直接获取类型信息
  - C++ 的四大扩展之一（其他三个是异常、名字空间和模板）

• 知识扩展

- dynamic_cast 安全向下转型
  - 向下转型，是指将基类指针（或引用）转换为派生类指针（或引用）
  - 语法： dynamic_cast< 目的类型 >(表达式)
  - dynamic_cast 与 static_cast 的不同点在于它是运行时转换符
  - 转换是有条件的
    - 如果指针（或引用）所指对象的实际类型与转换的目的类型兼容，则转换成功进行；
    - 否则如执行的是指针类型的转换，则得到空指针；如执行的是引用类型的转换，则抛出异常。

• 知识扩展

– 在 vc++ 中， RTTI 选项默认是关闭的，需要在 Project|settings|C/C++| Category 中选 C++ Language| 勾选 Enable Run-Time Type Information[RTTI]

```
// 注意打开 vc++ 的 RTTI 选项
#include <iostream>
using namespace std;
class A
{
  public:
    virtual void f1()
      {
        cout<<"A 类中的 f1 调用 "<<endl;
      }
    virtual ~A(){ } // 确保基类是多态类
};
```

- 知识扩展

```
class B:public A
{
  public:
    virtual void f1()
    {  cout<<"B 类中的 f1 调用 "<<endl; }
    virtual void f2()
    {  cout<<"B 类中新增的 f2 调用 "<<endl; }
};
class C:public B
{
  public:
    virtual void f1()
    {  cout<<"C 类中的 f1 调用 "<<endl; }
    virtual void f2()
    {  cout<<"C 类中的 f2 调用 "<<endl; }
};
```

知识扩展

```
void frame(A *pA)
{
  pA->f1();
  if (B *pB=dynamic_cast<B*>(pA) )// 将转换与结果测试写在同一句中，避免未经测试就使用
    { pB->f2(); }
  else
    {cout<<" 转换失败 "<<endl; }
}
int main()   {
  A a,*pA;   B b;  C c;
  cout<<"---A 类指针指向 A 类对象，强转为 B 类指针 ---"<<endl;
  pA=&a;   frame(pA);
  cout<<"---A 类指针指向 B 类对象，强转为 B 类指针 ---"<<endl;
  pA=&b;   frame(pA);
  cout<<"---A 类指针指向 C 类对象，强转为 B 类指针 ---"<<endl;
  pA=&c;   frame(pA);     return 1;
}
```

• 知识扩展

- 运行结果：

```
---A 类指针指向 A 类对象，强转为 B 类指针 ---
A 类中的 f1 调用
转换失败
---A 类指针指向 B 类对象，强转为 B 类指针 ---
B 类中的 f1 调用
B 类中新增的 f2 调用
---A 类指针指向 C 类对象，强转为 B 类指针 ---
C 类中的 f1 调用
C 类中的 f2 调用
Press any key to continue
```

- 知识扩展

- 用 typeid 获取运行时类型信息
  - 语法形式： typeid (表达式) 或 typeid (类型说明符)
  - 功能：

    返回一个对应于该类型的 type_info 类型的常引用，以描述对象的确切类型。

    - 表达式有多态类型时，会被求值，并得到动态类型信息；
    - 否则，表达式不被求值，只能得到静态的类型信息。

    类型信息用 type_info 对象表示

    - 为所有的内置类型和多态类型的对象保存运行时类型信息，其定义在头文件 < typeinfo > 中

• 知识扩展

- C++ 标准只规定了 type_info 必需提供以下四种操作，对具体的实现方式没有明确限定，因此不同编译器实现方式可能不同。
  （1） t1== t2 如果两个对象 t1 和 t2 类型相同，则返回 true，否则返回 false 。
  （2） t1 != t2 如果两个对象 t1 和 t2 类型不同，则返回 true，否则返回 false 。
  （3） t.name() 返回类型的名称，是一个 C 风格的字符串。
  （4） t1.before(t2) t1 出现在 t2 之前，返回 true，否则返回 false 。

• 知识扩展

- 【】 typeid 示例

```
// 注意打开 vc++ 的 RTTI 选项
#include <iostream>
//#include <typeinfo>
using namespace std;
class A
{
  public:
    virtual void f1()
      {
        cout<<"A 类中的 f1 调用 "<<endl;
      }
    virtual ~A(){ }
};
```

- 知识扩展

```
class B:public A
{
  public:
    virtual void f1()
    {  cout<<"B 类中的 f1 调用 "<<endl; }
    virtual void f2()
    {  cout<<"B 类中新增的 f2 调用 "<<endl;}
};
class C:public B
{
  public:
    virtual void f1()
    {  cout<<"C 类中的 f1 调用 "<<endl; }
    virtual void f2()
    {  cout<<"C 类中的 f2 调用 "<<endl;}
};
```

• 知识扩展

```
void frame(A *pA)
{
  const type_info &t1=typeid(pA);
  cout<<"typeid(pA):"<<t1.name()<<endl;
  const type_info &t2=typeid(*pA);
  cout<<"typeid(*pA):"<<t2.name()<<endl;
  if (t2==typeid(B))
  {
    cout<<" 是 B 类 "<<endl;
    static_cast<B*>(pA)->f2();
  }
  else
    cout<<" 不是 B 类 "<<endl;
}
```

- 知识扩展

```
int main()  {
  A a,*pA;   B b;  C c;
  cout<<"---A 类指针指向 A 类对象，指针的类型与所指对象的类型 ---"<<endl;
  pA=&a;
  frame(pA);
  cout<<"---A 类指针指向 B 类对象，指针的类型与所指对象的类型 ---"<<endl;
  pA=&b;
  frame(pA);
  cout<<"---A 类指针指向 C 类对象，指针的类型与所指对象的类型 ---"<<endl;
  pA=&c;
  frame(pA);
  return 1;
}
```

• 知识扩展

- 运行结果：

```
---A 类指针指向 A 类对象，指针的类型与所指对象的类型 ---
typeid(pA):class A *
typeid(*pA):class A
不是 B 类
---A 类指针指向 B 类对象，指针的类型与所指对象的类型 ---
typeid(pA):class A *
typeid(*pA):class B
是 B 类
B 类中新增的 f2 调用
---A 类指针指向 C 类对象，指针的类型与所指对象的类型 ---
typeid(pA):class A *
typeid(*pA):class C
不是 B 类
Press any key to continue
```

因为精确比较，所以没能调 f2，不够灵活

小结

1、运算符重载

通过定义运算符重载函数可以实现运算符的重载，即对于用户定义的类的对象，可以使用系统定义的运算符。运算符重载函数可以是成员函数或友元函数。一般将双目运算符重载为友元函数，单目运算符重载为成员函数。由于友元的使用会破坏类的封装性，因此从原则上说，要尽量将运算符函数作为成员函数。

2、重载为成员函数与友元函数的区别

如果将运算符重载作为成员函数，由于它可以通过 this 指针自由访问本类的数据成员，因此可以少写一个函数的参数。但必须要求运算表达式第一个参数 (即运算符左侧的操作数) 是一个类对象，因为必须通过类的对象去调用该类的成员函数。而且重载函数的返回值与该对象类型相同，只有运算符重载函数返回值与该对象同类型，运算结果才有意义。

小结

3、不同类型数据间的转换

对于标准类型的转换，编译系统有章可循，知道怎样进行转换。而对于用户自己声明的类型，编译系统并不知道怎样进行转换。需要定义转换构造函数实现将一个其它类型的数据转换成一个类的对象，定义类型转换函数将一个类的对象转换成其它类型的数据。

4、多态性的概念

从系统实现的角度看，多态性分为两种：静态多态和动态多态。以前学过的函数重载和运算符重载实现的多态性属于静态多态性，在程序编译时系统就能决定调用的是哪个函数，因此静态多态性又称为编译时的多态性。静态多态性是通过函数的重载实现的（运算符重载实质上也是函数重载）。动态多态是在程序运行过程中才动态地确定操作所针对的对象。它又称运行时的多态性。动态多态是通过虚函数实现的。

小结

5、虚函数的定义和使用方法

在基类中由 virtual 声明成员函数为虚函数。虚函数的作用是允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问派生类中的同名函数。

6、纯虚函数和抽象类的定义

在虚函数的定义中不仅有 virtual，还有 =0，表示该函数是一个纯虚函数。包含纯虚函数的类是一个抽象类，不能定义抽象类的对象。

7、面向对象程序设计的基本思想

面向对象的程序具有以下的两个特性：界面的继承性。这里派生类的界面继承了基类界面的某些特性；实现的继承。派生类可以使用从基类继承的某些特性从而简化派生类的设计。

感谢观看