第一章：C++基础

抛出异常

```
throw <表达式>;
```

注意：当一个异常被一个函数引发后，执行流程返回到该函数的调用者中。

```
double Div(double a,double b){

      if (b==0.0) throw 1.0E-38;  return a/b;

    }
```

检查异常

```
 try{
   可能发生异常的语句;
   }
```

注意：

（1） **try**块的语句在执行过程中可能会发生多种类型的异常，由**catch**块进行捕获并处理；

（2） 异常发生后，执行流程立即转到**catch**块，执行异常处理。

异常捕获并处理

```
catch (类型 变量1){

      异常处理语句1;

    }


    catch (<参数声明n>){

      异常处理语句n;

    }
```

注意：（1）每个**catch**块声明了所要捕获的一种类型。

（2）**catch**语句块可以很多，但是最多只有一个**catch**语句块被执行。

带默认参数的函数

按从左到右的次序将实参和形参结合

```
 int add(int x,int y=10);

//      add(15);  ->    add(15,10);
```

注意:默认形参值必须由右向左的顺序定义。

如果某个参数有默认值，则其右面的参数必须都有默认值；如果某个参数没有默认值，则其左面的参数都不能有默认值。

```
int  max(int a, int b=10, int c=20);            //正确

int  max(int a, int b=10, int c);                  //错误

int  max(int a=5, int b, int c=30);              //错误

//在后两种情况下，调用语句 x = max(20, 30); 会出错!
```

注意2：在使用带默认参数值的函数时，只能在函数定义或函数声明中的一个位置给出默认值，不能在两个位置同时给出。----不能重复给出默认值

注意3：还要保证在函数调用之前给出默认值。

注意4:一个函数既是重载函数，又带默认参数，调用中会可能会出现二义性。

```
int add(int x=5, int y=6) {return x+y; }
float add(int x=5, float y=10.0) { return x+y; }
int main() {

int a; float b;       a= add(10,20);

b= add(10);     //本句产生二义性,重载函数的第一个参数是int型无法区分

return 0;
}
```

引用:实质是被引用变量(对象)的地址

引用是变量的别名。

引用与其“引用”的变量**公用一个存储空间**，不单独占用存储空间，并且引用的类型和其“引用”的变量的类型保持一致。

```
类型说明符  &引用名 = 变量名;

int  a = 1;     int  &b = a;//引用的类型和原变量的类型保持一致
//b是a的别名,定义b的同时就要进行初始化
```

注意:函数不能返回对局部变量(对象)的引用

函数内部初始化的局部变量(对象),在函数结束后就可能被销毁了,因此不能返回

内联函数 inline

内联函数与一般函数的区别在于它不是在调用时发生控制转移，而是在编译时将被调函数体嵌入到每一个函数调用处，节省了参数传递、控制转移等开销。

内联函数的定义

```
inline  类型说明符  函数名（参数及类型表{
函数体
}
    inline int add(int x,int y,int z)
    {
      return x+y+z;
}
```

```
inline int f(int x){return x*x;}
int main(){
int x(2);
cout<<f(x)<<;”;
cout<<f(x+1)<<endl;
return 0;
}
/*
程序运行结果:
4;9
原因:
f(x)  替换为  2*2
f(x+1)  替换为  3*3
    inline相比于define,在语句之间多了括号
 */
 //-------------------------------------------
#define f(x) x*x
int main(){
    int x(2);
```

```
    cout<<f(x)<<“;”;
    cout<<f(x+1)<<endl; return 0;
}
/*
程序运行结果：
4;5
原因：
f(x)     替换为□2*2
f(x+1) 替换为□2+1*2+1
*/
```

作用域运算符

直接用在全局函数前，表示是全局函数。当类的成员函数跟类外的一个全局函数同名时，在类内定义的时候，打此函数名默认 调用的是本身的成员函数；如果要调用同名的全局函数时，就必须打上::以示区别。

```
float a = 13.5;

int main(){

int a = 5;

std::cout<<a<<std::endl;        // 输出局部变量

std::cout<<::a<<std::endl;     //::a 输出全局变量

return 0;

}
```

动态分配/撤销内存

new操作符动态分配内存建立的变量，称为“堆对象”或者“在动态存储中分配”。new运算符创建的堆对象一直存在，直到使用delete运算符显示的销毁。

```
new <类型说明符> (<初始值列表>)
int *p1 = new int(5);
delete p1;
int *p2 = new int[5];
delete [] p2;    //销毁动态创建的数组
```

注意: **new**运算符返回一个与**new**所分配对象类型相匹配的指针； 如果**new**运算符不能分配到所需要的内存，将返回**0**，这时为空指针。

使用**new**运算符创建对象时，它可以根据其参数来选择适当的构造函数；

使用**new**创建对象数组时，不能为该数组指定初始值，其初始值为缺省值。所以，使用**new**创建对象数组时，类中必须说明缺省构造函数或者带默认值的构造函数。

访问修饰符

访问修饰符	访问范围	继承性
private	本类内部	不可继承
default	本类+同包	同包子类可以继承
protected	本类+同包+子类	可以继承
public	公开	可以继承

类也可以由struct关键字声明，strust与class的区别是： 如果不指定访问权限，**struct**缺省的访问权限是公有的，而**class**是私有的。

第二章：类和对象

类的成员函数

内联成员函数

将成员函数的定义直接写在类中即成为内联成员函数，或者在类外定义时用inline指出。如：

```
inline void CRect::SetColor(string)
{
   color = c;
}  //内联成员函数
```

带默认参数值的成员函数

```
void SetSize(int l=100, int w=100);//声明

void CRect::SetSize(int l=100, int w=100){
        length=l; width = w;
    }   //定义

//默认参数只能在声明或定义中的一处给出,二者选一
```

构造函数

特点:

1. 构造函数的函数名与类名相同。
2. 不能定义构造函数的类型（即不能指明构造函数返回值的类型）。

3. 构造函数应声明为公有函数。
4. 构造函数不能在程序中调用，在对象创建时，构造函数被系统自动调用。

如果类中没有构造函数，则系统会自动生成函数体为空的默认构造函数。

拷贝构造函数

用一个已经存在的对象初始化新对象，拷贝构造函数的**参数为该类对象的引用**。

```
类名::类名(const 类名&  引用名)
point::point(const point& p);
```

注意：

每个类都必须有一个拷贝构造函数；[重要] 如果类中没有说明拷贝构造函数，则编译系统自动生成一个具有上述形式的默认拷贝构造函数，依次完成类中对应数据成员的拷贝。

使用拷贝构造函数的三种情况

1. 明确表示由一个对象初始化另一个对象时； 例如：TPoint N(M);
2. 当对象作为函数实参传递给函数形参时（传数据值调用）； 例如：P=fun(N);
3. 当对象作为函数返回值时（数据值）； 例如：return R;

析构函数

在删除一个对象前被调用，释放该对象成员的内存空间，以及其它一些清理工作。

特点:

1. 析构函数名字为符号"~"加类名。
2. 析构函数没有参数，不能指定返回值类型。
3. 一个类中只能定义一个析构函数，所以析构函数不能重载。
4. 当一个对象作用域结束时或者使用delete运算符释放new运算符创建的对象时，系统自动调用析构函数。

如果类中没有提供析构函数，系统会自动生成一个函数体为空的析构函数。

注意:

先创建的对象后调用析构函数 后被创建的对象先调用析构函数 简称为"先构造后析构，后构造先析构"

类的组合[类充当成员变量]

构造函数的执行顺序[先子对象后本身]

1. 按子对象的声明顺序**[父对象的构造函数实参列表里子对象的顺序]**依次调用子对象的构造函数。
2. 然后执行组合类本身的构造函数体。

注意：构造函数的初始化列表中未给出对子对象的初始化，则表示使用子对象的缺省构造函数初始化子对象；

析构函数的执行顺序[先本身后子对象]

1. 执行组合类本身的析构函数体。
2. 然后按子对象的声明**[父对象的构造函数实参列表里子对象的顺序]**相反的顺序依次调用子对象的析构函数。

```
int main() {
        CPoint p1(1, 1), p2(4, 5);
        CLine l(p2, p1);        //p1和p2的顺序,此时p2的构造先调用
        cout << "The distance is :";
        cout << l.GetDistance() << endl;
        system("pause");
        return 0;
}
```

静态成员

静态数据成员

特点:在每个类中只有一个拷贝，由该类的所有对象共同维护和使用。

```
static 类型标识符 静态数据成员名;
static int m;   //定义
类型标识符 类名::静态数据成员名 = 初始值;
int point::m = 15;      //初始化
类名::静态数据成员名
point::m        //使用
```

静态数据成员不属于任何一个对象，可以通过类名直接对它进行访问

静态成员函数

作用：只能操作静态成员。

公有的静态成员函数，可以通过类名或对象名来调用

```
cout << CStudent::GetTotal() << endl;
//即使未定义CStudent类的对象，也可以访问静态数据成员total了
```

常成员【记】

常数据成员

```
const 类型说明符 数据成员;
const int m;
```

注意：

1. 常数据成员必须被初始化，并且不能被更新；
2. 一般情况下，常数据成员只能通过初始化表的方法进行初始化；

```
const int a;  const int& r;

A::A(int i):a(i),r(a){} //通过初始化表初始化常数据成员r
```

常成员函数

```
类型  函数名（参数表) const;
int function() const;
```

注意:

1. const是函数类型的一个组成部分，在函数实现部分必须带有**const**关键字。

```
void Print() const {cout << X <<", "<< Y << endl;}
//const不能省
```

2. 常成员函数中不能改变类数据成员的值。

```
class point{
public:
int x
int getValue(int a) const        //参数列表可以有非常成员
{
    x = a;       //错误，不能更改类的数据成员的值
    a = 5;       //正确，a是在函数作用域内定义的
   return x;
}
};
```

3. 常成员函数不能直接调用非常成员函数。

```
int getValue() const
{
   return w*h + getValue2();//getValue2()非常成员函数
   //错误的不能调用其他非常成员函数。
}
```

4. const关键字可以用于对**重载函数的区分**。

5. 参数列表可以有非常成员。

常类型【记】

一般常类型

注意：

1. 常类型的变量或对象的值是不能被更新的；
2. 定义或说明常类型时必须初始化；

```
类型说明符 const 常量名（初始值）；
const 类型说明符 常量名（初始值）；
int const x=2;
const int x=2;
int const a[3]={1,2,3};
const int a[3]={1,2,3};
```

常指针

1. 指针常量**【常指针】** 指针是个常量，即 指针的地址值不能改变。

```
char * const ps1 = s1;  //const修饰ps1，地址不能变
```

2. 常量指针 指向常量的指针，即所指向的值不能改变。

```
const char  *ps2=s2;   //const修饰char这个整体，值不能变
```

注意： const的位置，用于区别指针常量与常量指针。

对象数组与对象指针

```
DATE dates[5]={ //构造函数的参数不止一个
DATE(7,22,1998),
DATE(7,23,1998),
DATE(7,24,1998)
};      //缺少的数组元素用默认构造函数进行初始化

dates[0]=DATE(7,22,1998);
dates[1]=DATE(7,23,1998);
dates[2]=DATE(7,24,1998);
//数组元素通过匿名对象赋值;不会调用拷贝构造函数
//DATE d1(7,22,1998);dates[0]=d1;
```

```
//释放d1

DATE dates[5]=(1,2,3,4);//构造函数的参数只有一个
```

对象指针略。

this指针

类的每个**非静态成员函数**都有一个隐含的this指针参数，不需要显示说明。

指向数据成员的指针【记】

定义：

```
<类型说明符>  <类名>:: *<指针名>
```

例如，定义一个指向类A的数据成员c的指针pc：

```
int A::*pc=&A::c;
```

指向成员函数的指针【记】

定义：

```
<类型说明符>  (<类名>:: *<指针名>)(参数表)
```

例如，定义一个指向类A的成员函数fun的指针pfun：

```
int (A::*pfun)(int)=&A::fun;     //记得加括号和取地址符&
A a;
(a.*pfun)(2);    //调用也要加括号
```

总结：用法上比普通指针多了域前缀

友元

友元函数

定义友元函数时，只要在函数原型前加入关键字friend，并将函数原型放在类中，格式为：

```
friend  类型标识符  友元函数名（参数列表）;
friend int function();
```

友元函数可以是一个普通函数，也可以是其他类的成员函数， 在其函数体中可以通过**对象名直接访问这个类的私有成员**。

友元类

友元类

如果一个类（如类A）的很多成员函数都需要经常访问另一个类（如类B）的私有成员，可以将类A声明为类B的友元。

若A类为B类的友元类，则A类的所有成员函数都是B类的友元函数，都**可以访问B类的私有成员**。**

第三章：类的继承与派生

类成员的访问权限：

1. **私有成员(private)：**可以被类自身的成员和友元访问，但不能被包括派生类在内的其他任何类和任何普通函数访问。
2. **公有成员(public)：**可以被任何普通函数和任何类的成员函数访问。
3. **保护成员(protected)：**可以被类自身的成员和友元访问外，还可以被派生类的成员函数访问。

继承方式【基类和派生方式取交集】

公有继承的特点：

1. 基类的私有成员不能被派生类的成员函数访问。
2. 基类的公有成员和保护成员在派生类中的访问权限不变。

保护继承的特点：

1. 基类的私有成员不能被派生类的成员函数访问。
2. 基类的公有成员和保护成员在派生类中的访问权限变为保护类型。

私有继承的特点：

1. 基类的私有成员不能被派生类的成员函数访问。
2. 基类的公有成员和保护成员在派生类中的访问权限变为私有类型。

派生类的构造过程和析构过程

基类的**构造函数和析构函数都不被继承**，需要在派生类中重新定义派生类的。 在初始化时，也**要同时初始化基类成员**，通过**调用基类的构造函数**。

```
派生类名::构造函数名(参数表):基类名(参数表),子对象1(参数表1),子对象2(参数表2),…,常量1(初值1),常量2(初值2),…,引用1(变量1),引用2(变量2)…{

派生类构造函数体;

}
Student1::Student1(int a,int b,int c,int,d):Student(a,b,c)
```

```
//调用基类构造函数初始化基类成员
{
    date = d;    //函数体中对派生类新增数据成员舒适化
}
```

构造执行顺序：

1. 先调用基类的构造函数;
2. 然后对派生类的成员初始化列表中数据成员（包括子对象、常量、引用等必须初始化的成员）按照声明顺序，依次初始化;
3. 最后执行派生类构造函数的函数体;

注意：

1. 构造函数的执行顺序**只与成员声明的顺序有关**，而与初始化表中各项的排列顺序无关。

```
class B : public A{
private:
int   b,c;
const int d;
A x,y;
public:
B(int v) : b(v), y(b+2), x(b+1), d(b), A(v){
c = v;
}       //先调用基类A(v),然后A x(b+1),然后A y(b+2),b(v),c(v)
~B(){}
};
```

2. **常数据成员、引用成员、子对象，只能在成员初始化列表中进行初始化。**

析构过程

派生类析构函数执行时将**自动调用基类及子对象的析构函数**，因此不必显式调用。

派生类析构的执行顺序：

1. 先执行派生类的析构函数体。
2. 然后按着子对象声明的相反顺序，依次调用子对象的析构函数。
3. 最后调用基类的析构函数。

多继承

```
class CDerived : public CBase1,private CBase2 {
};//基类构造函数的执行顺序是从左到右，先1后2
```

其余与单继承一致

二义性

基类有同名成员引起的二义性

多继承时，**不同基类可能有同名成员**，这样派生类中就可能有从不同基类继承的同名成员，在引用时产生二义性。

从多个路径继承同一个基类引起的二义性

多重继承时，低层的派生类有可能从不同的路径**继承同一个基类的成员多次**，引用这样的成员时也会产生二义性。

解决方法

1. 使用域运算符和基类名来限定【不同基类同名成员】

```
d1.CBase1::GetB0();
d1.CBase2::GetB0();
```

2. 将间接基类声明为虚基类【不同路径继承同一个基类】 虚基类的成员在它的间接派生类中只被继承一次

```
class DBase::virtual public CBase1,public CBase2{

};
```

注意:

1. 派生类中只有一个虚基类子对象；
2. 虚基类构造函数必须只被调用一次，目的是要保证虚基类子对象只被初始化一次；
3. 最远派生类：继承结构中建立对象时所指定的类；
4. 虚基类子对象由最远派生类的构造函数通过调用虚基类的构造函数进行初始化；
5. 在一个成员初始化列表中出现对虚基类和对非虚基类构造函数的调用时，虚基类的构造函数先于非虚基类的构造函数的执行；
6. 最远派生类的构造函数的成员初始化列表中必须给出对虚基类的构造函数的调用；如果未列出，则相应的虚基类必须有缺省构造函数。(如1,2,3) 虚基类中定义了含参数的构造函数但没有默认构造函数，则他的所有派生类都要给他初始化(如1.2.3.4)

```
class A{
    A (int i){};//1.含参数的构造函数，但是没有默认构造函数
}
class B:virtual public A{
    B(int n):A(n){};//2.初始化A
```

```
};
class C:virtual public A{
    C(int n):A(n){};//3.初始化A
};
class D:public B,public C{      //D是派生链中最后一个类
    D(int a,int b,int c):A(a),B(b),c(c){};
};//4.在初始化D的直接基类BC的同时，还要初始化虚基类，也就是A
int main(){
    D d1(1,2,3);//定义D类，D类就是最远派生类
}
```

最远派生类

```
class CBase0 {
protected:
        int b0;
public:
        CBase0(int x = 0) {
                b0 = x; cout << "CBase0构造函数调用"<< endl;
        }
        int GetB0() { return b0; }
};

class CBase1 : virtual public CBase0 {

};

class CBase2 : virtual public CBase0 {

};

class CDerived : public CBase1, public CBase2 {

};

int main() {
        CDerived d1(10, 15, 20);//定义了CDerived类，他就是最远派生类
        cout << d1.GetB0() << endl;     //定义谁，建立谁的子对象，谁就是
        cout << d1.CBase1::GetB0() << endl;
        cout << d1.CBase2::GetB0() << endl;
        system("pause");
        return 0;
}       //
```

多继承中：派生类的构造函数和析构函数的执行顺序

1. 按定义顺序自左至右地执行【所有虚基类】的构造函数；
2. 按定义顺序执行派生类的【所有直接基类】的构造函数；
3. 按定义顺序派生类构造的【所有数据成员，包括子对象、const成员和引用成员】；
4. 执行派生类自身的构造函数体。

派生类对象的析构顺序与构造顺序相反。

第四章：多态

静态多态：程序在编译时即可确定名称其含义的多态。

动态多态：程序在执行时才可确定名称其含义的多态；

按照性质，多态可分为：重载、泛型(或模板)、多态变量和重写。

1. 重载：函数重载和运算符重载。
2. 泛型（模板）：函数模板和类模板。

运算符重载

不允许重载的运算符

```
.   (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）
```

运算符被重载为类的成员函数

```
 <类型> operator <运算符>(<参数表>){ …… }
complex operator +(const complex& c);//声明
```

1. 重载为类的成员函数时，隐含了一个参数(this指针)。
2. 重载一元运算符，不再显式说明参数。 为区分前置和后置运算符，C++规定：对于前置单目运算符，重载函数没有参数；对于后置单目运算符，重载函数有一个整型参数，这个整型参数没有其他用途，只是用于区分前置运算与后置运算。

   ```
   complex operator ++();  //++a;前置单目运算符
   complex operator ++(int);       //a++;后置单目运算符
   ```

3. 重载二元运算符，只显式说明一个参数。 该参数为操作数的右操作数，左操作数由this指针（指向调用该成员函数的对象）提供。

   ```
   complex operator +(const complex& c);
   //c1+c2 -> c1.operator + (c2)
   ```

其他类型重载运算符

```
int& operator [](int i){        //下标运算符
    return v[i];
};      //返回值为引用类型的重载
double F::operator () (double x,double y) {
    return x*y+5;
}       //函数调用运算符
```

return this* 返回的是当前对象的克隆或者本身 若返回类型为A**， 则是克隆 若返回类型为**A&**， 则是本身

return this 返回当前对象的地址（指向当前对象的指针）

运算符重载为类的友元函数

```
friend <类型> operator <运算符>(<参数表>){}
friend complex operator + (const complex& c1,const complex& c2){
    double r=c1.real+c2.real;
    double i=c1.imag+c2.imag;
    return complex(r,i);
}
//
```

1. 重载为友元函数时，没有隐含的参数**this**指针，即不改变原有运算符的语法结构。
2. 下列运算符不能重载为友元函数：=、()、[]、->

赋值运算符重载

当类中有指针类型的数据成员时，必须定义拷贝初始化和赋值操作 否则编译器生成的拷贝初始化操作和赋值操作的执行将导致程序在运行时产生问题。

```
class A
{
public:
    A(int i);
    ~A();
private:
     int *p;
};

A::A(int i)
{
    p=new int(i);
}
A::~A()
{
```

```
    delete p;
}

int main(){
A a(5);
A b(a); //用a对象给b赋值，指针p指向同一对象，析构两次
A c(5),d(10);
d=c;    //c的p指针赋值给d的p指针，d原来的数据将无法访问，且析构两次
}

//------加入以下
A(A& r){
        p=new int(*r.p);
        cout<<"copy complete!"<<endl;
    }   //拷贝构造函数
A& operator =(A& r){
        if(this==&r)  return *this;
        *p = *r.p;   cout<<"evaluation complete!"<<endl;
        return *this;
    }   //赋值操作符重载
```

静态联编与动态联编

赋值兼容原则

如果派生类的**继承方式为public**，则这样的**派生类**称为基类的**子类**，而相应的**基类**则称为派生类的**父类**。

假设B类是A类的子类

1. 可将B类的对象赋值给A类的对象。**子类可给父类赋值**

```
A a = b;
```

2. 可将B类对象的地址值赋给指向A类对象的指针。**子类地址可给父类地址赋值**

```
B* b = new B;
A* a = b;
```

3. 可将B类对象赋值给A类对象的引用。**子类可给父类对象的引用赋值**

```
A& a = b;
```

静态联编

在编译时进行的绑定，即**编译时就确定了**程序中的操作调用与执行该操作代码之间的关系。

通过静态联编就可确定某一个多态的具体含义，则该多态为静态多态。

```
void fun(A& d){
    cout<<d.Geta()*10<<endl;
}       //编译时就确定了执行的是A类中的Geta()方法
```

动态联编

在运行时进行的绑定，即**运行时才能确定**了程序中的操作调用与执行该操作代码之间的关系。

通过动态联编所确定某一个多态的具体含义，则该多态为动态多态。

C++动态联编在虚函数的支持下实现。

虚函数

```
virtual <类型> <函数名> (参数列表);
virtual int function(int a);
```

注意：虚函数不能是static成员函数。

若类中一成员函数被说明为虚函数，则该成员函数在子类中可能有不同的实现。 当使用该成员函数操作指针或引用所标识的对象时，对该成员函数调用可采用动态联编方式，即在程序执行时根据指针所指向（引用所引用）的对象的类型来调用该成员函数。

```
int main(){
    CShape *ps[3];
    CShape s("Red");
    CPoint p1(10,10), p2(100,100),p3(50,50);    //继承了CShape
    CLine  l(p1,p2,"Green");
    CCircle c(p3, 20, "Black");                                    //继承了CShape
    ps[0] = &s;   ps[1] = &l;  ps[2] = &c;
    for(int i=0; i<3; i++) ps[i]->Draw();       //虽然都是CShape类型数组，输出的结果不同
    return 0;
}
```

注意：如果采用普通对象来调用虚函数，则将采用静态联编方式执行。

1. 将成员函数声明为虚函数，在函数原型前加关键字virtual，如果成员函数的定义直接写在类中，也在前面加关键字virtual。
2. 派生类继承基类的**继承方式必须为公有继承**。
3. 将成员函数声明为虚函数后，再将**基类指针（父类指针）指向派生类对象（子类对象）**，在程序运行时，就会根据**指针指向的具体对象所属的类来调用各自的虚函数**，称之为动态多态。
4. 如果基类的成员函数是虚函数，在其**派生类中，原型相同的函数自动成为虚函数**。

> 一个类中若有虚函数，（不论是自己的虚函数，还是继承而来的），那么类中就有一个成员变量：虚函数指针，这个指针指向一个虚函数表，虚函数表的第一项是类的typeinfo信息，之后的项为此类的所有虚函数的地址。
>
> 假设经过成员对齐后的类的大小为size个字节。那么类的sizeof大小可以这么计算：size + 4*n（虚函数指针的个数n）。
>
> 带有虚函数的类的sizeof大小，实际上和虚函数的个数不相关，相关的是虚函数指针。

虚函数的重写

若基类（父类）中一成员函数被说明为虚函数，在派生类（子类）中定义重新给出该虚函数的定义，称为重写。

子类重写父类中的虚函数，必须满足**子类中的虚函数的原型和父类中的虚函数的原型一致**，即满足如下规则：

1. 子类虚函数的参数个数必须和父类虚函数的个数相同。
2. 子类虚函数的参数的类型必须和父类虚函数的参数的类型均相同。
3. 子类虚函数的返回类型必须和父类虚函数的返回类型相同。

构造函数和析构函数中调用虚函数

构造函数和析构函数中调用虚函数时，采用**静态联编**，即**构造函数调用的虚函数是自己类中定义的虚函数**，如果自己类中没有实现这个虚函数，则**调用基类中的虚函数**，而不是任何派生类中实现的虚函数。

虚析构函数

如果基类的析构函数定义为虚析构函数，则**派生类的析构函数**就会**自动成为虚析构函数**。

```
class A
{
public:
~A() {cout<<"A::~A() called. "<<endl;}
};
class B:public A{
public:
    B(int i) {buf=new char[i];}
        ~B(){
        delete [] buf;
        cout<<"B::~B() called. "<<endl;
    }
private:
    char *buf;
};

A *a = new B(10);
delete a;       //此时调用的只有基类A的析构函数而不会调用B的析构函数
//将析构函数都定义成虚函数就可以解决
```

纯虚函数

纯虚函数是**不必定义函数体**的特殊虚函数

```
virtual 函数类型 函数名（参数表）= 0;
virtual int function() = 0;
```

抽象类

含有纯虚函数的类称为抽象类。抽象类常常**用作派生类的基类**。

如果派生类继承了抽象类的纯虚函数，却**没有重新定义原型相同且带函数体的虚函数**，或者**派生类定义了基类所没有定义的纯虚函数**，则派生类仍然是抽象类，在多层派生的过程中，如果到某个派生类为止，**所有纯虚函数都已全部重新定义**，则该派生类就成为非抽象类。

注意：抽象类没有对象，但是可以有指针和引用

第五章：Template

注意：不是任何数据类型都可以参数化，是否可以参数化决定于运算符。

函数模板

```
template<模板参数表>
返回类型 函数名(参数表)
{
//函数体
}
template <typename T, typename S>
T add(T a, S b){
    return  a + b;
}
```

辨析：函数模板与模板函数

1. 函数模板是对一组函数的描述，不是函数
2. 模板函数是编译器在程序中发现有与函数模板参数表相匹配的函数调用时而生成的重载函数。
3. 模板函数是一种实实在在的函数

类模板

```
template<模板参数表>
class 类名{
//类声明体
};
template<typename T>
class A{
public:
```

```
    A();
    T pop();//声明
    void push(const T&);
};
//成员函数的定义
template<typename T>
T A<T>::pop(){
    //函数体
}
template<typename T>
void A<T>::push(const T& temp){
    //函数体
}
```

类模板中的所有成员函数均为函数模板。

类模板的说明部分和实现部分应该在同一个文件中。