考点

2019年12月31日　　20:49

C++的核心：封装、接口、类、对象

面向对象三大特性：抽象、继承、多态

析构函数与构造函数的名字相同，没有参数，没有返回值，当撤销对象时，被编译器自动调用。

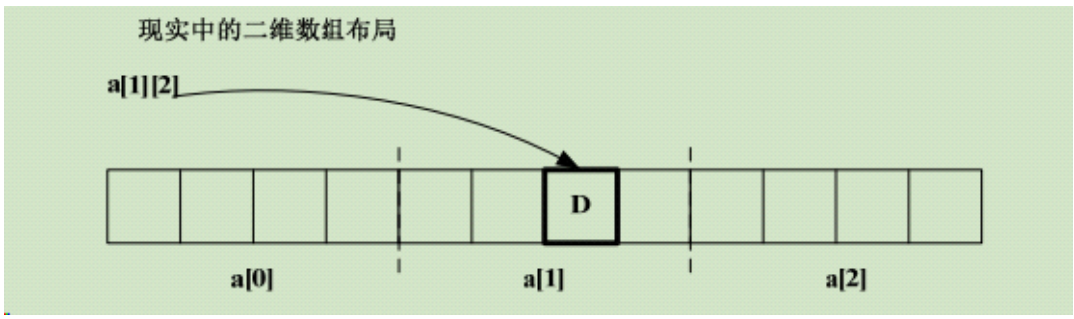
```
int a[10];//求数组元素个数
```

(利用sizeof函数求得对应的数)

样例代码：

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    int a[3];
    cout << sizeof(a) / sizeof(int) << endl;
    return 0;
}
```

二维数组的储存：编译器总是将二维数组看成是一个一维数组，而一维数组的每一个元素又都是一个数组。



一维数组和二位数的初始化

```
char a[] = "aaa";
char a[3] = "aaa";///这个是错的。 “aaa”真正的长度是4个单位，因为真是的“aaa”其实是这样的aaa\0末尾还有一个\0需要占位
```

```
D:\编程\C++\测... 15     error: initializer-string for array of chars is too long [-fpermissive]
```

动态内存分配数组：

一维数组创建方式：

```
int a[10];
///初始化 int c[5] = {0,1,2,3,4};
```

```
int* b;
b = new int[10];///创建连续的一块内存,可通过b[1]这样的访问方式调用
```

内存释放：delete []b;

二维数组的创建：

```
int a[2][3] = {1,2,3,4,5,6};
///或者 int a[2][3] = {{1,2,3},{4,5,6}};

int b[][3] = {1,2,3,4,5,6};
///或者 int b[][3] = {{1,2,3},{4,5,6}};、

int** group;
group = new (int*)[10];
for(int i = 0 ; i < 10 ; i++)
    group[i] = new int[10];
///这样可以创建一个group[10][10];10*10的数组 可以通过 group[i][j];使用

int x,y;
cin>>x>>y;
int **a = new int*[x];
a[0] = new int[x*y];
```

类对象数组类似于

```
class Point{
public:
    int a;
    int b;
private:
    Point(int _a,int _b)
    {
        a = _a;
        b = _b;
    }
    Point()
    {
        a = 0;
        b = 0;
    }
    ~Point()
    {}
};
Point b[2] = {Point(1,2),Point(2,3)};
```

可以借助构造函数进行初始化，如果没有参数传入，则直接调用无参数的构造函数。

对象指针 可以通过类似于结构体的“->”方式调用

如：

```
#include <iostream>
using namespace std;
class Point
{
public:
    Point();
    ~Point();
private:
};
int main()
```

```
{
    Point* a;
    a->~Point();
    (*a).~Point();///另一种写法
    return 0;
}
```

（Ps：在类内有一个“this”指针，这个指针指向当前类本身可以通过上面的方法直接访问类内成员）

new int(10)是分配一个int，用（）中的10初始化。
new char[10]是分配10个int，没有初始化。
如果是类的话，new会调用类的默认构造函数。

清空new创建的内存：

一般变量内存直接用　Delete 变量名 释放内存

如果是数组这用 delete[] 指针名（高维也是如此）；

高维数组创建：

例子：

```
float (*cp)[25][10];
cp = new float[10][25][10];
/*
创建高维数组可以预先创建一个差一维的指针型数组；
然后分配用户需要的维度空间大小；
*/
```

可以通过类似数组的方式访问，也可通过指针偏移的方法访问

（关于深复制与浅复制：在使用指针作为对象，且用到指针与复制时要用深拷贝，因为直接进行指针的赋值，相当于拷贝地址，共享地址的变量。几个同时指向一块地址的对象指针，他们当中的任何一个被释放都会导致其他几个指针无法访问该内存。深复制，即以新建内存的方式，逐条复制，在一块新的内存上创建一块内容一样的内存。）

带默认形参值的函数

即：在函数创建的时候在形参括号内，变量在一开始就用等号添加初始值

例子：

```
int add(int x,int y = 5,int z = 6);//正确
int add(int x = 1,int y = 5,int z);//错误
int add(int x = 1,int y,int z = 6);//错误
```

创建过程中：必须从右向左赋值，中间不能有隔一个之类的情况（调用顺序是从左向右）

在相同作用域内，不允许 在同一个函数的多个声明中对同一个参数的默认值进行重复定义->即使前后定义值相同也不行

```
#include <iostream>
using namespace std;
int add(int x = 1,int  y = 5);
int main()
{
    int b;
    b = add();
    cout << b << endl;
```

```
    return 0;
}
int add(int x = 1, int y = 5)
{
    int z;
    z = x * y;
    return z;
}
```

正确的格式（这里把原函数中的赋值语句注释了）：

```
#include <iostream>
using namespace std;
int add(int x = 1,int  y = 5);
int main()
{
    int b;
    b = add();
    cout << b << endl;
    return 0;
}
int add(int x /*= 1*/, int y /*= 5*/)
{
    int z;
    z = x * y;
    return z;
}
```

三种派生方式的概念：

公有继承（public）：公有成员和保护成员访问属性不变出现在子类中，而私有成员不可直接访问。

私有继承（private）：公有成员和保护成员访问属性变为私有成员出现在子类中，而私有成员不可直接访问。

保护继承（protect）：公有成员和保护成员访问属性变为保护成员出现在子类中，而私有成员不可直接访问。

STL容器的概念:()

vector访问方便，List插入删除方便，查找某个元素慢；set(关联容器，set中每个元素的值都唯一，而且系统能根据元素的值自动进行排序。)&map查找快、插入&删除开销大

this指针概念：每一个普通成员函数都有，指向当前对象。

数组创建方法：

一维数组创建方式：

```
int a[10];
```

```
///初始化 int c[5] = {0,1,2,3,4};
int* b;
b = new int[10];///创建连续的一块内存,可通过b[1]这样的访问方式调用
```
内存释放：delete []b;

二维数组的创建：

```
float(*fp)[25][10];
cp = new float[10][25][10];
delete [] cp;

int a[2][3] = {1,2,3,4,5,6};
///或者 int a[2][3] = {{1,2,3},{4,5,6}};

int b[][3] = {1,2,3,4,5,6};
///或者 int b[][3] = {{1,2,3},{4,5,6}};、

int** group;
group = new (int*)[10];
for(int i = 0 ; i < 10 ; i++)
    group[i] = new int[10];
///这样可以创建一个group[10][10];10*10的数组 可以通过 group[i][j];使用

int x,y;
cin>>x>>y;
int **a = new int*[x];
a[0] = new int[x*y];
```

类对象数组类似于

```
class Point{
public:
    int a;
    int b;
private:
    Point(int _a,int _b)
    {
        a = _a;
        b = _b;
    }
    Point()
    {
        a = 0;
        b = 0;
    }
    ~Point()
    {}
};
Point b[2] = {Point(1,2),Point(2,3)};
```

可以借助构造函数进行初始化，如果没有参数传入，则直接调用无参数的构造函数。

对象指针 可以通过类似于结构体的“->”方式调用

如：

```
#include <iostream>
using namespace std;
class Point
{
public:
    Point();
```

```
    ~Point();
private:
};
int main()
{
    Point* a;
    a->~Point();
    (*a).~Point();///另一种写法
    return 0;
}
```

（Ps：在类内有一个“this”指针，这个指针指向当前类本身可以通过上面的方法直接访问类内成员）

new int(10)是分配一个int，用（）中的10初始化。
new char[10]是分配10个int，没有初始化。
如果是类的话，new会调用类的默认构造函数。

清空new创建的内存：

一般变量内存直接用　Delete 变量名 释放内存

如果是数组这用 delete[] 指针名（高维也是如此）；

高维数组创建：

例子：

```
float (*cp)[25][10];
cp = new float[10][25][10];
/*
创建高维数组可以预先创建一个差一维的指针型数组；
然后分配用户需要的维度空间大小；
*/
```

可以通过类似数组的方式访问，也可通过指针偏移的方法访问

（关于深复制与浅复制：在使用指针作为对象，且用到指针与复制时要用深拷贝，因为直接进行指针的赋值，相当于拷贝地址，共享地址的变量。几个同时指向一块地址的对象指针，他们当中的任何一个被释放都会导致其他几个指针无法访问该内存。深复制，即以新建内存的方式，逐条复制，在一块新的内存上创建一块内容一样的内存。）

内联函数

关键字 Inline（放在函数开头）

发生在编译预处理（用于取代宏）

{

调用该函数相当于调用该函数的位置用该函数代替

主要格式

在编译时嵌入每一个调用处

优点：可以节省参数传递、控制转移等开销；

Inline 数据类型 函数名（参数）

{

主要语句

}

样例：

```
inline double calArea(double radius)
```

```
        {
                return PI*radius*radius;
        }
        int main()
        {
                double r = 3.0;
                double area = calArea(r);
                cout << area << endl;
                return 0;
        }

}
```

类数组:

```
class C;  C c[3];
```

引用的本质:

```
int* a;
int b;
a = &b;///这里的引用符号的意义是取地址，获取变量的
int add(int& a);///这里的引用此时函数中对形参的各种操作实际上是对实参本身进行操作，而非简单的将实参变量或对象的值拷贝给形参
///即：直接传参数本身
```

普通变量的存储

不能重载的运算符有哪些

栈&队列的基本操作

```
stack<int> stk;
queue<int> que;
int a;
///stk 栈的基本操作
stk.push(a);///入栈，将一个元素压入栈顶
stk.pop();///出栈操作，无返回值
```

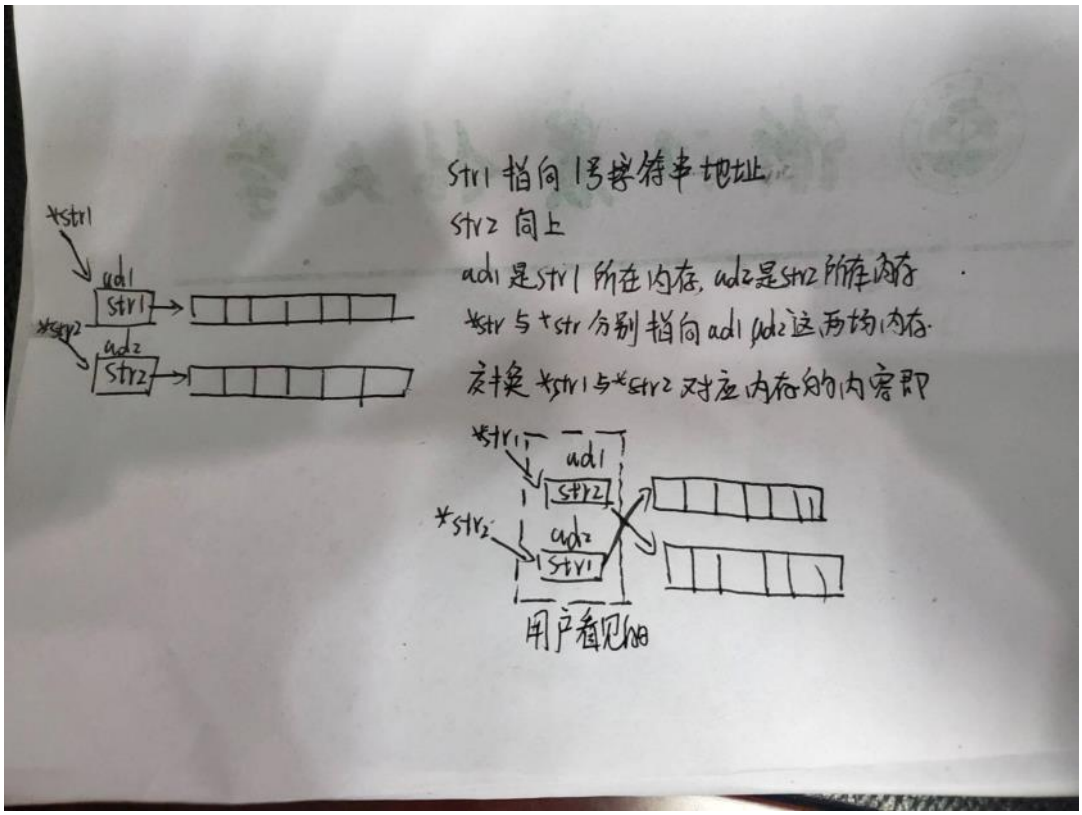
```
stk.top();///访问栈顶的元素
stk.size();///返回栈内元素的个数

///que 队列的基本操作
que.push(a);///入队操作，将一个元素放入队尾
que.pop();///将一个队首元素弹出
que.front();///访问队首的元素
que.size();///返回队列的元素个数
```

传指针或引用的交换字符串的swap()

指针法：

```
#include <string.h>
#include <iostream>
#include <string>
using namespace std;
int main()
{
     ///二级指针传入直接交换地址
    void swap(char** num1, char** num2);
    char* s1 = (char*)"qwert";
    char* s2 = (char*)"asdfghjkl";
    cout << "交换前：s1 = " << s1 << ", s2 = " << s2 << endl;
    swap(&s1, &s2);
    cout << "交换后：s1 = " << s1 << ", s2 = " << s2 << endl;
    return 0;
}
void swap(char** num1, char** num2)
{
    char* t;
    t = *num1;
    *num1 = *num2;
    *num2 = t;
}
```



引用法：

```
#include <string.h>
#include <iostream>
#include <string>
using namespace std;
int main()
{
    void swap(char*&num1, char*&num2);
    char* s1 = (char*)"qwert";
    char* s2 = (char*)"asdfghjkl";
    cout << "交换前：s1 = " << s1 << ", s2 = " << s2 << endl;
    swap(s1, s2);
    cout << "交换后：s1 = " << s1 << ", s2 = " << s2 << endl;
    return 0;
}
///Char*是一个整体表示字符串指针，引用则是直接传入对应的保存了地址的指针变量然后交换 下面
void swap( char*& num1, char*& num2 )
{
    char* t;
    t = num1;
    num1 = num2;
    num2 = t;
}
```

sizeof的使用(变量，字符串，对象等)

```
///变量的大小取决于对应数据类型的大小，字符串大小取决于开辟的变量区域大小，对象的大小取决于内部变量大小
///的总和
#include <iostream>
#include <fstream>
#include <stack>
#include <queue>
using namespace std;
class Point
{
```

```
public:
    Point();
    ~Point();
    int add(int _a, int _b)
    {
        s1 = _a;
        s2 = _b;
    }
private:
    int s1, s2, s3;
};
Point::Point()
{
}
Point::~Point()
{
}
int main()
{
    int Int;
    char str[10];
    cout<<"int "<< sizeof(Int) << endl;
    cout<<"char str[10] "<< sizeof(str) << endl;
    cout << "sizeof class " << sizeof(Point) << endl;
    return 0;
}
```

传值和传引用与函数的结合

```
int add(int a,int b);///传参数，与原参数无关，仅仅对原数据进行一个复制
int add(int& a,int& b);///传引用，直接将原参数传入，函数内的操作会引起传入的参数发生改变
```

list的引用

```
void push_front(const T & val) //将 val 插入链表最前面
void pop_front()   //删除链表最前面的元素
void sort()     //将链表从小到大排序
void remove (const T & val)     // 删除和 val 相等的元素
remove_if(begin,end,op);   //  删除符合某种条件的元素
void unique()   //删除所有和前一个元素相等的元素,传入迭代器“iterator”
void merge(list <T> & x)     //将链表 x 合并进来并清空 x。要求链表自身和 x 都是有序的
void splice(iterator i, list <T> & x, iterator first, iterator last)   // 在位置 i 前面插入链表 x 中的区间 [first, last)，并在链表 x 中删除该区间。链表自身和链表 x 可以是同一个链表，只要 i 不在 [first, last) 中即可
```

静态成员变量的应用

静态成员：

静态数据成员（static）：（！！只初始化一次）（P155）

static：是描述该类的所有对象共同特征的一个数据项，对于任何对象实例，他的属性值是相同的；

静态函数成员具有静态生存期。不属于任何一个对象，因此可以通过类名对它进行访问，一般的用法是

“类名::标识符”

```
#include <iostream>
using namespace std;
class Point{
     public:
          Point(int x = 0 ; int y = 0):x(x),y(y){
              count++;
          }
          Point(Point &p){
              x = p.x;
              y = p.y;
              count++;
          }
          ~Point()
          {count--;}
          int getx()
          {
              return x;
          }
          int gety()
          {
              return y;
          }
          void showcount()
          {
              cout << " object count = " << count << endl;
          }
     private;
         int x,y;
         static int count;
};
int Point::count = 0;
int main()
{
    return 0;
}
```

静态的成员函数可以直接访问该类的静态数据和函数成员。而访问非静态成员，必须通过对象名。

简单的模板函数

主要格式:template<typename T> 函数类型呢名 函数名（参数表）；

参照样例即可

```
#include <iostream>
#include <fstream>
#include <stack>
#include <queue>
#include <string>
#include <string.h>
using namespace std;
template<typename T>
T add(T a, T b)
{
    T c;
    c = a + b;
    return c;
}
int main()
```

```
{
    int a, b ,c;
    a = 1;
    b = 5;
    c = add<int>(a, b);
    cout << c << endl;
    return 0;
}
```

虚基类:

将共同基类设置为虚基类，这时从不同的路径继承过来来的同名数据成员在内存中就只有一个副本，同一个函数名也只有一个映射。

格式:

```
class 派生类名:virtual 继承方式 基类名
```

样例对比:

```
#include <iostream>
using namespace std;
class Base0{
    public:
        int var0;
        void fun0(){cout<<"Member of Base0"<<endl;}
};
class Base1:public Base0{
    public:                                 //新增外部接口
        int var1;
};
class Base2:public Base0{
    public:                                 //新增外部接口
        int var2;
};
class Derived:public Base1,public Base2{
    public:                                 //新增外部接口
        int var;
        void fun(){cout<<"Member of Derived"<<endl;}
};
int main(){
    Derived d;
    d.var0=2;
    d.fun0();
    return 0;
}
```

程序显然无法成功运行，错误原因：`request for member 'var0' (fun0) is ambiguous (不明确的)`，因为Base1和Base2都有`var0`和`fun0()`，程序不知道你调用的是哪一个类继承下来的，除非你用作用域分辩符来唯一标识

这样写:

```
d.Base1::var0=2;
d.Base1::fun0();
d.Base2::var0=3;
d.Base2::fun0();
d.Base1::var0=2;
d.Base1::fun0();
d.Base2::var0=3;
d.Base2::fun0();
```

正确的写法：

```
#include <iostream>
using namespace std;
class Base0{
    public:
        int var0;
        void fun0(){cout<<"Member of Base0"<<endl;}
};
class Base1:virtual public Base0{
    public:                                    //新增外部接口
        int var1;
};
class Base2:virtual public Base0{
    public:                                    //新增外部接口
        int var2;
};
class Derived:public Base1,public Base2{
    public:                                    //新增外部接口
        int var;
        void fun(){cout<<"Member of Derived"<<endl;}
};
int main(){
    Derived d;
    d.var0=2;                          //直接访问虚基类的数据成员
    d.fun0();                          //直接访问虚基类的函数成员
    return 0;
}
```

这时的Derived类中，通过Base1，Base2继承来的基类Base0中成员var0和fun0只有一份副本，改变var0的值，就相当于改变Base1，Base2中的var0的值了。

模板类运算符的重载

重载的样例：

```
#include <iostream>
#include <fstream>
using namespace std;
template<class T>
class Group
{
public:
    Group();
    ~Group();
    Group(T* _a);
    Group(Group<T>& _a);
    void disp();
    Group<T>& operator = (Group<T>& _new);
    bool operator == (Group<T>& _new);
    T& operator [] (int index);
    Group<T>& operator ++ ();
    Group<T>& operator ++ (T);
    Group<T>& operator+(Group<T> _new);
    Group<T>& operator+(T* _new);
     ///这里friend函数必须写在类内，类外声明会出错。
    friend ostream&  operator << (ostream& out, Group<T>& _new)
    {
        for (int i = 0; i < 5; ++i)
            out << _new.head[i];
```

```
        return out;
    }
private:
    T head[5];
    /// = , == , ++ , -- , + ,- ,[] , << , >>
};
template<class T>
Group<T>::Group()
{
    for (int i = 0; i < 5; ++i)
        head[i] = 0;
}
template<class T>
Group<T>::~Group()
{
}
template<class T>
Group<T>::Group(T* _a)
{
    for (int i = 0; i < 5; ++i)
        head[i] = _a[i];
}
template<class T>
Group<T>::Group(Group<T>& _a)
{
    for (int i = 0; i < 5; ++i)
        head[i] = _a.head[i];
}
template<class T>
void Group<T>::disp()
{
    for (int i = 0; i < 5; i++)
        cout << head[i] << ' ';
    cout << endl;
}
template<class T>
Group<T>& Group<T>::operator = (Group<T>& _new)
{
    for (int i = 0; i < 5; i++)
        this->head[i] = _new.head[i];
    return *this;
}
template<class T>
bool Group<T>::operator == (Group<T>& _new)
{
    for (int i = 0; i < 5; ++i)
    {
        if (this->head[i] != _new.head[i])
            return false;
    }
    return true;
}
template<class T>
Group<T>& Group<T>::operator+(Group<T> _new)
{
    static Group <T> nk;
    for (int i = 0; i < 5; ++i)
    {
```

```
        nk.head[i] = this->head[i] + _new.head[i];
    }
    return nk;
}
template<class T>
Group<T>& Group<T>::operator+(T* _new)
{
    static Group<T> nk;
    for (int i = 0; i < 5; ++i)
        nk.head[i] = this->head[i] + _new[i];
    return nk;
}
template<class T>
T& Group<T>::operator [] (int index)
{
    if (index >= 0 && index < 5)
        return this->head[index];
    exit(0);
}
template<class T>
Group<T>& Group<T>::operator ++ ()
{
    for (int i = 0; i < 5; ++i)
        ++(this->head[i]);
    return *this;
}
template<class T>
Group<T>& Group<T>::operator ++ (T)
{
    static Group<T> old = *this;
    for (int i = 0; i < 5; ++i)
        ++(this->head[i]);
    return old;
}


int main()
{
    int b[5] = { 1,2,3,4,5 };
    int c[5] = { 1,1,1,1,1 };
    Group<int> g1(b);
    Group<int> g2(c);
    g1.disp();
    g2.disp();
    Group<int> g3 = g1 + g2;
    g3.disp();
    Group<int> g4 = g1 + g2 + g3;
    g4.disp();
    ++g4;
    g4.disp();
    Group<int> g5 = g4++;
    g5.disp();
    cout << g5[2] << endl;
    cout << g5 << endl;
    return 0;
}
```

虚函数：必须是非静态成员函数，虚函数经过派生之后，在类族中就可以实现运行过程中的多态。简而言之就是用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。这种技术可以让父类的指针有

“多种形态”，这是一种泛型技术。

（虚函数在生成过程中，会在目标类的内存最前面生成一个虚函数表内存，记录虚函数的地址）
主要格式：virtual 函数类型 函数名（形参列表）；
虚函数声明只能出现在类定义中的函数原型声明中，而不能在成员函数实现的时候。

这些函数不能成为虚函数；
1.内联函数：我们都知道内联函数只是在函数调用点将其展开，它不能产生函数符号，所以不能往虚表中存放，自然就不能成为虚函数。
2.静态函数：定义为静态函数的函数，这个函数只和类有关系，它不完全依赖于对象调用，所以也不能成为虚函数。
3.构造函数：都知道只有当调用了构造函数，这个对象才能产生，如果把构造函数写成虚函数，这时候我们的对象就没有办法生　成。更别说用对象去调用了。所以构造函数不能成为虚函数。

针对书上函数声明后面加const
并不是因为是虚函数才要加上const，而是因为它是一个不会修改数据成员的函数。为了避免不慎将数据成员作修改，或者调用了其他非const成员函数，所以才加上const的
（const后面加一个“=0”则成为纯虚函数，基类就可以不再给出函数的实现部分）

```
#include <iostream>
#include <fstream>
using namespace std;
class Base1
{
public:
    virtual void display() const;
private:
};
void Base1::display() const {
    cout << "Base1::display()" << endl;
}
class Base2:public Base1
{
public:
    void display()const;
private:
};
void Base2::display()const {
    cout << "Base2::display()" << endl;
}
class Derived:public Base2
{
public:
    void display() const;
private:
};
void Derived::display()const
{
    cout << "Derived::display()" << endl;
}
void fun(Base1* ptr)
{
    ptr->display();
}
```

```
int main()
{
    Base1 base1;
    Base2 base2;
    Derived derived;
    fun(&base1);
    fun(&base2);
    fun(&derived);
    return 0;
}
#include <iostream>
#include <fstream>
using namespace std;
class Base1
{
public:
    virtual void display() const;
private:
};
void Base1::display() const {
    cout << "Base1::display()" << endl;
}
class Base2:public Base1
{
public:
    void display()const;
private:
};
void Base2::display()const {
    cout << "Base2::display()" << endl;
}
class Derived:public Base2
{
public:
    void display() const;
private:
};
void Derived::display()const
{
    cout << "Derived::display()" << endl;
}
void fun(Base1* ptr)
{
    ptr->display();
}
int main()
{
    Base1 base1;
    Base2 base2;
    Derived derived;
    fun(&base1);
    fun(&base2);
    fun(&derived);
    return 0;
}
```