

1. 面向对象的程序设计方法具有 3 大特征：封装、继承、多态。
2. 面向对象的编程语言具有 4 个主要特征：抽象、封装、继承、多态。

a.抽象：

是指有意忽略问题的某些细节和与当前任务无关的方面，用对象的某些特征属性来代表该对象。如，仅使用姓名和年龄来代表一个人(类后加分号)：

```
class Person
{
    char name[20];
    int age;
};
```

b.封装：

是指把描述一个对象的属性和行为的代码封装在一个模块中，也就是一个类中。属性(attribute)用变量定义，行为(behavior)用方法定义，方法可以直接访问所属对象的属性。例如，一个 time 类中，包含的属性为时分秒，包含的方法为时分秒的设置：

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

class Time
{
private:
    int hour, minute, second;
public:
    void setTime(int h, int m, int s);
    void setHour(int h) {hour = h;}
    void setMinute(int m) {minute = m;}
    void setSecond(int s) {second = s;}
    void display() {cout << hour << ":" << minute << ":" << second << endl;}
};

void Time::setTime(int h, int m, int s)
{
    hour = h;
    minute = m;
    second = s;
}

int main()
{
    Time T;
    T.setTime(12, 53, 24);
    T.display();
}
```

c.继承：

是指在已有类的基础上，添加新的属性或者方法。继承分为单继承和多继承，单继承是指仅从一个基类派生；多继承是指从多个基类派生。C++支持单继承和多继承。

d.多态：

多态性指的是相同的操作、函数或过程可作用于多种类型的对象上并获得不同的结果。典型的代表为函数重载，即“同一接口，多种方法(多种实现)”。函数重载时，判据为参数个数以及参数类型，对函数返回值不做检查：

```

void add(int a, int b)
{
    cout << "int型 : " << a + b << endl;
}

void add(double a, double b)
{
    cout << "double型 : " << a + b << endl;
}

int main()
{
    int a = 1, b = 1;
    double c = 1, d = 1;

    add(a, b);
    add(c, d);

    return 0;
}

```

```

int型 : 2
double型 : 2

-----
Process exited after
请按任意键继续. . .

```

e.消息:

是指对象之间在交互通信中所传送的信息，即一个对象对另一个对象发出的请求。消息分为公有(public)和私有(private)两类。调用对象中的函数(方法)就是向该对象传送一个消息(message)，要求该对象实现某种行为(功能)。

3. 类与对象:

类(class)是一种抽象数据类型，是具有相同属性(attribute)和相同行为(behavior)的对象(object)的集合。对象是一个实体，是类的一个实例。对象的两个必备要素为属性和行为，属性是静态特征，行为是动态特征。

对象的抽象是类，类的实例是对象。比如定义了 Person 类，Amy 是 Person 类的一个实例，也就是一个对象：

```

class Person
{
private:
    string name;
    int age;
public:
    void setName(string &p_name) {name = p_name;}
    void setAge(int p_age) {age = p_age;}
    void display(){cout << "Name : " << name << "\nAge : " << age << endl;}
};

int main()
{
    Person Amy;
    string p_name = "Amy";
    Amy.setName(p_name);
    Amy.setAge(15);
    Amy.display();

    return 0;
}

```

```

Name : Amy
Age : 15

```

4. const 与 define:

const 与 define 都可以用来定义常量，但是它们有着明显的区别：const 定义的常量是一个**常变量**，占存储单元，有地址，可以使用指针指向该常量，默认为 int 型；define 定义的是符号**常量**，没有类型，不是变量，不占据内存单元，仅仅是一个标识符，在预编译时进行字符置换。

变量：值可以改变，分配内存空间

常量：值不可改变，不分配内存空间

常变量：值不可改变，分配内存空间

```
#define PI 3.1415926  
  
const double PI = 3.1415926;
```

上面两种定义方式均可。

5. 变量引用(reference):

引用就是给变量赋予一个新的名字，可以通过这个别名来间接地引用该变量，对引用变量的操作代表了对该变量的操作，**引用与原始的变化是一致的**。

变量引用不会分配新的内存空间，别名和原名对应的变量地址一样，因此二者的值是一样的。

声明方式：**类型 &别名 = 原名**；或者是 **类型& 别名 = 原名**；

变量引用在声明时**必须初始化**。

```
#include<iostream>  
  
using namespace std;  
  
int main()  
{  
    int a;  
    int &b = a;  
  
    cout << "a的地址 : " << &a << "\nb的地址 : " << &b << endl;  
  
    return 0;  
}
```

```
a的地址 : 0x6ffe14  
b的地址 : 0x6ffe14
```

6. 函数重载:

运算符重载:

运算符也是对象，也可以重载。在类中，运算符重载的声明方式为：

返回类型 operator 运算符(类型 右操作数地址)；

在类外定义时，先用上述语句在类内声明，然后使用下面的方式定义：

类名:: 返回类型 operator 运算符(类型 右操作数地址) { //函数体; }

例如，定义一个复数类，将“+”运算符重载，使得它可以完成复数运算：

```

class complex
{
private:
    double real, imag;
public:
    complex(double r, double i);
    complex operator +(complex &right);
    void display() {std::cout << real << "+" << imag << "i" << std::endl;}
};

complex::complex(double r=0, double i=0)
{
    real = r;
    imag = i;
}

complex complex::operator +(complex &right)
{
    return complex(real + right.real, imag + right.imag);
}

int main()
{
    complex a(1, 1), b(1, 1), c;
    c = a + b;
    c.display();

    return 0;
}

```



```
2+2i
```

注意，如果构造函数中带有参数，那么在创建对象时一定要传入对应参数。上面的 complex 类采用了默认参数初始化的方式，因此可以直接使用 complex c 来定义，此时不要加括号，complex c() 是错误的。

模板函数：

模板函数可以简化函数重载，使得我们不需要写多个除了函数。模板函数不要求明确定义参数类型，而是使用虚拟类型 T，声明方式如下：

```

template<typename T> //这里不要加分号
T 函数名(T 参数 1, T 参数 2, T 参数 3, ……) { //函数体; }

```

```

template<typename T>

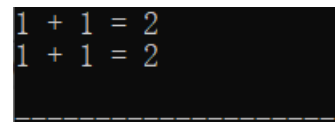
T add(T a, T b)
{
    cout << a << " + " << b << " = " << a + b << endl;
}

int main()
{
    int a = 1, b=1;
    double c = 1, d = 1;

    add(a, b);
    add(c, d);

    return 0;
}

```



```
1 + 1 = 2
1 + 1 = 2
```

7. 内联函数

内联函数(inline function)又叫做内置函数,可以提高效率。通常在出现函数调用时,操作系统需要保存现场,转去执行被调用的函数,然后恢复现场,继续执行,这些操作将会耗费大量时间。

内联函数在编译时,会嵌入到**主函数**中,提高运行效率,减少了调用。声明方式如下:

inline 返回值类型 函数名(参数表) { //函数体; }

模板函数也可以设置为 inline 函数。

```
template<typename T>

inline T add(T a, T b)
{
    cout << a << " + " << b << " = " << a + b << endl;
}

int main()
{
    int a = 1, b=1;
    double c = 1, d = 1;

    add(a, b);
    add(c, d);

    return 0;
}
```

```
1 + 1 = 2
1 + 1 = 2
```

8. 字符串变量:

字符串变量即 string 类,使用时需要包含头文件, #include<string>。

字符串可以直接赋值,也可以拷贝赋值。字符串可以使用“+”进行连接。

```
#include<iostream>
#include<string>

using namespace std;

int main()
{
    string s1, s2, s3;
    s1 = "Hello ";
    s2 = "World !";
    s3 = s1 + s2;
    cout << s3 << endl;
    return 0;
}
```

```
Hello World !
```

9. 动态分配内存与撤销: new 与 delete

new 可以用来动态申请新的内存分配,使用完后,可以用 delete 清除内存占用。

这与 C 语言中的 malloc() 和 free() 作用类似。值得注意的是,用 new 申请的内存空间不会自动释放,需要使用 delete 手动释放。使用方法如下:

指针变量 = new 类型; //p 为分配的内存的首地址, *p 为存储的内容

delete 指针变量;

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

int main()
{
    while(true)
    {
        int *p;
        p = new int;
    }

    return 0;
}
```

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

int main()
{
    while(true)
    {
        int *p;
        p = new int;
        delete p;
    }

    return 0;
}
```

左侧的操作将会使得内存很快被全部占用，右侧的则会正常工作。因此，在使用 new 申请动态分配内存空间时，在使用结束后一定要用 delete 释放掉。

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

int main()
{
    int *p;
    p = new int;

    *p = 10;
    cout << p << "处的值为 : " << *p << endl;

    delete p;
    cout << p << "处的值为 : " << *p << endl;

    return 0;
}
```

```
0x151530处的值为 : 10
0x151530处的值为 : 1382960
-----
```

在申请新空间的时候可以赋值：

```
int *p; //声明指针变量
p = new int(10); //申请新内存的同时赋初始值 10
delete p; //释放申请的内存空间
```

如果是**数组**，则可以使用下面的声明方式：

```
int *p; //声明指针变量
p = new int[20]; //申请长度为 20 的整型数组内存空间
delete []p; //释放数组占据的内存空间，delete p 仅仅释放数组的第一个位置
```

10. 类与结构体：

类的声明使用 class 关键词，而结构体的声明使用 struct 关键字。在不对属性进行声明的情况下，struct 默认为 public，class 默认为 private。因此，类与结构体相比，最大的不同在于增加了**信息的隐蔽性**，除此之外，二者没有其他区别。

11. 类成员访问控制：

类成员的访问权限被限制为 public、protected、private 三种。

a. public:公有成员，类内类外均可访问。

- b. protected:保护成员，只能类内访问，派生类也可以访问。
- c. private:私有成员，只能类内访问，友元函数可以访问。

```
class Account
{
    private:
        string account_name;
        double balance;

    public:
        Account(string, double);
        void info();
};

Account::Account(string name="Default", double remain=0)
{
    account_name = name;
    balance = remain;
}

void Account::info()
{
    cout << "Account : " << account_name << "\nBalance : " << balance << endl;
}

int main()
{
    Account my_account;
    my_account.info();
}
```

```
Account : Default
Balance : 0
-----
```

首先，注意构造函数是没有返回值的，也不能增加返回值类型。另外，在上图中，账户名称与账户余额为私有变量，如果我们使用 `my_account.account_name` 和 `my_account.balance` 是无法访问的，但是借助于公有函数 `info()`，我们可以获取到账户的信息。因此，通过类的公有函数可以实现对类的私有成员和保护成员的访问。

由于不同的类具有不同的作用域，因此，不同类的成员可以同名。

不能在类的声明中给类成员赋初值，只有在类的对象定义之后才可以对数据成员赋值，因此，在创建对象时，会调用类的构造函数。

12. 类的成员函数：

类的成员函数其实就是类的对外接口，因此，通常将它们定义为 `public` 类型。类的成员函数可以在类内定义，也可以在类外定义，在类内定义时无需声明，在类外定义时，必须声明。

为了防止类的定义过于冗长，通常将类的成员函数定义在类的外部，定义方式为：返回值类型 类名::函数名(参数表){//函数体;} 在类内应当进行声明，声明格式为：public/protected/private 返回值类型 函数名(参数表);

如下例，在类内定义构造函数，在类外定义显示函数：

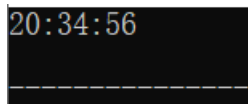

```

class Time
{
    private:
        int hour, minute, second;
    public:
        Time(int h=0, int m=0, int s=0) {hour=h; minute=m; second=s;}
        void display();
};

void Time::display()
{
    cout << hour << ":" << minute << ":" << second << endl;
}

int main()
{
    Time T(20, 34, 56);
    T.display();
    return 0;
}

```



同样的，由于类在调用成员函数时也需要保存现场，执行调用，恢复现场。当调用函数频繁时，也会带来效率的降低。因此，可以将类的成员函数设置为内联函数 (inline function)。内联成员函数的声明有两种方式：隐式声明和显示声明。

隐式声明：不需要使用 inline 关键字，直接将成员函数定义在类的内部。**在类内定义的成员函数都被隐式声明为内置(内联)成员函数。**

显式声明：需要使用 inline 关键字。适用于成员函数定义在类的外部情况。只需在返回值类型前加 inline 关键字即可，类内的函数声明无需改变。格式如下：

```
inline void Time::display() { //函数体; }
```

13. 对象：对象是类的实例化

类是对象的抽象集合，对象是类的具体实例。在创建对象之前，必须先定义类。对象的创建主要有三种方式：（注意，类的花括号后要加分号）

a. 先声明类，然后定义对象

```

class Time
{ //类的结构 };
Time t1, t2; //创建对象

```

b. 声明类的同时定义对象

```

class Time
{ //类的结构 } t1, t2; //同时创建对象

```

c. 不出现类名，直接定义对象(很少使用)

```

class
{ //类的结构 } t1, t2; //不出现类名，直接创建对象

```


14. 对象成员的引用方式:

对象成员的引用也主要有三种方式:

- 通过对象名和成员运算符(“.”)访问:
- 通过指向对象的指针和“->”访问, 等价于(*p).成员名
- 通过对象的引用变量进行访问

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

class Person
{
public:
    string Name;
    Person(string name="Default"){Name = name;}
};

int main()
{
    Person me("Amy");
    Person *p = &me;
    Person &ref = me;
    cout << "a : " << me.Name << "\nb : " << p->Name << "\nc : " << ref.Name << endl;

    return 0;
}
```

```
a : Amy
b : Amy
c : Amy
```

上图中的 p->Name 等价于(*p).Name

15. 对象的自引用指针: this

当创建一个类的若干对象后, 每个对象拥有自己的对象成员(属性), 但是, 所有对象的成员函数是共享的, 只有一份。因此, 在调用成员函数时, 必须要明确是哪一个对象在调用, 只有这样, 才能正确的使用该对象的属性值。在 C++ 中, 上述要求是通过 this 指针实现的。

成员函数拥有一个名为 this 的指针, 这个指针为自引用指针。每当创建一个对象时, 系统将指向新创建的对象(首地址)的指针作为隐含的参数传递给成员函数。因此, 不同的对象调用相同的成员函数时, 根据成员函数的 this 指针确定是哪一个对象在调用。

```

#include<iostream>
using namespace std;

class Number
{
private:
    int number;
public:
    Number(int temp=0){number = temp;}
    void SetNumber(int new_number) {number = new_number; cout << this << endl;}
};

int main()
{
    Number N1, N2;
    N1.SetNumber(20);
    cout << &N1 << endl;

    cout << "\n";

    N2.SetNumber(30);
    cout << &N2 << endl;

    return 0;
}

```

```

0x6ffe10
0x6ffe10

0x6ffe00
0x6ffe00

```

从输出结果可以看出，N1 的 this 指针与 N1 的首地址是一致的，N2 的 this 指针与 N2 的首地址是一致的。通常情况下，this 指针是隐含存在的，具体的使用方法后续会介绍。

16. 构造函数与析构函数：

构造函数(constructor)：

类的数据成员不能在声明类时初始化。原因在于，类并不是一个实体，而仅仅是一种类型，是抽象的，不占用内存空间，当然也就无法容纳初始数值。

构造函数的主要用途是执行对象数据成员的初始化操作(当然，其他操作也是可以的)，并且，构造函数不需要用户调用，在创建对象时系统自动调用(用户无法调用构造函数)，这种调用是自动的隐式调用，在创建对象时立即调用。

构造函数应当是公有的，否则对象无法调用。

构造函数不具有任何类型，不具有返回值。

构造函数是与类名相同的公有成员函数。

构造函数可以重载，可以带有默认参数。当未定义构造函数时，C++自动生成一个构造函数，该构造函数不执行任何操作，没有参数。

构造函数重载：

构造函数重载与普通的函数重载一致，即可以定义多个参数个数或参数类型不同的构造函数，从而适应不同情况下的初始化操作，实例如下：

```

#include<iostream>
using namespace std;

class Time
{
private:
    int hour, minute, second;
public:
    Time(){hour = 0; minute = 0; second = 0;}
    Time(int h, int m, int s){hour = h; minute = m; second = s;}
    void display(){cout << hour << ":" << minute << ":" << second << endl;}
};

int main()
{
    Time t1, t2(10, 10, 10);
    t1.display();
    t2.display();

    return 0;
}

```

```

0:0:0
10:10:10

```

上述样例中对构造函数进行了重载，不带参数时将默认赋值为 0，带参数时将会使用带参构造函数。

默认构造函数：

在调用构造函数时，**不必给出实参**的构造函数成为默认构造函数 (default constructor)。如果未定义构造函数，C++ 会默认提供一个构造函数，但是该构造函数不进行任何初始化操作。一个类只能有一个默认构造函数，这是因为，不带参数的构造函数如果有多个，编译器将无法执行重载，无法根据实参判断该调用哪一个构造函数。

默认构造函数分为两类，一是**无参数**的构造函数，二是带有**全部默认参数**的构造函数。

只要一个类定义了构造函数，C++ 将不再为其创建空构造函数。

下面是一个错误示例，**不能定义多个默认构造函数**：

```

class Time
{
private:
    int hour, minute, second;
public:
    Time(){hour = 0; minute = 0; second = 0;}
    Time(int h=10, int m=10, int s=10){hour = h; minute = m; second = s;}
    void display(){cout << hour << ":" << minute << ":" << second << endl;}
};

```

拷贝构造函数：对象的赋值/对象的复制/拷贝构造函数

a. 对象的赋值：

若对一个类定义了多个对象，则这些对象之间可以相互赋值。对象之间的赋值可以使用 “=” 运算符实现。例如：

```

class t1(10, 10, 10);
class t2;
t2 = t1; // 对象的赋值

```

b. 对象的复制:

有时,我们可能会用到多个完全相同的对象,这时,可以使用对象复制来快速生成若干个完全相同的对象。例如:

```
class t1(1, 2, 3);  
class t2 = t1; // 对象的复制  
class t3(t1); // 对象的复制
```

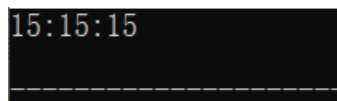
c. 拷贝构造函数:

拷贝构造函数是一种特殊的**构造函数**,其作用是用一个已经存在的对象去初始化一个新的同类对象。**拷贝构造函数的实参是一个已创建的同类对象**。拷贝构造函数的声明如下:

类名(类名 &对象名){//构造函数体;}

具体的实例如下,为 Time 类创建两个构造函数,一个是带全部默认参数的默认构造函数,一个是拷贝构造函数:

```
#include<iostream>  
using namespace std;  
  
class Time  
{  
private:  
    int hour, minute, second;  
public:  
    Time(int h=10, int m=10, int s=10){hour = h; minute = m; second = s;}  
    Time(Time &T);  
    void display(){cout << hour << ":" << minute << ":" << second << endl;}  
};  
  
Time::Time(Time &T)  
{  
    hour = T.hour + 5;  
    minute = T.minute + 5;  
    second = T.second + 5;  
}  
  
int main()  
{  
    Time t1;  
    Time t2(t1);  
    t2.display();  
    return 0;  
}
```



15:15:15

对象 t1 调用了带全部默认参数的构造函数,对象 t2 调用了拷贝构造函数。另外,当使用一个已创建的对象来初始化一个新的本类对象时,如果在类中没有定义拷贝构造函数,那么系统会自动生成一个默认的拷贝构造函数,类似于对象的复制。

参数初始化表:

除了上述常用的初始化方法,C++还提供了参数初始化表来对对象数据进行初始化,这种方法的目的是使代码更加简洁。例如:

```
Time::Time(int h, int m, int s): hour(h), minute(m), second(s) {}
```

参数列表后面要加冒号，赋值语句类似于对象复制，中间用逗号隔开，最后的花括号不能省略。

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

class Time
{
private:
    int hour, minute, second;
public:
    Time(int h, int m, int s): hour(h), minute(m), second(s) {}
    void display(){cout << hour << ":" << minute << ":" << second << endl;}
};

int main()
{
    Time t(23, 46, 52);
    t.display();

    return 0;
}
```



析构函数：

析构函数也是一种特殊的成员函数，访问权限为 public。析构函数的作用是释放分配给对象 (类的实例) 的内存空间，并做一些善后工作。当撤销对象时，系统将会自动调用析构函数。

析构函数的名称与类名相同，前加“~”波浪号。

析构函数同构造函数一样，没有返回值，没有类型。

析构函数没有参数，不能重载(因为没有参数)，一个类只能有一个析构函数。

函数重载依靠的是参数个数和参数类型之间的差异。没有参数必然无法重载。

析构函数的声明格式如下：

类名:: ~类名 () { //析构函数体; } // 定义在类外。定义在类内时，不需要“类名:: ”

// 注意，析构函数没有参数，参数表为空，但括号不可省略

在类中声明时，只需要~类名 ();即可，若为空析构函数，则花括号不可省略~类名 () {};

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

class Time
{
private:
    int hour, minute, second;
public:
    Time(): hour(0), minute(0), second(0) {}
    ~Time() {cout << "析构函数被调用" << endl;}
    void display() {cout << hour << ":" << minute << ":" << second << endl;}
};

int main()
{
    Time t;
    t.display();
    return 0;
}
```

```
0:0:0
析构函数被调用

-----

Process exited after 0.2959 seconds with return value 0
请按任意键继续. . .
```

上面的示例中使用了**不带参数的默认构造函数**，风格为**参数初始化表**，析构函数在 return 0 执行之前被系统自动调用。

需要注意的问题：

- a. 每个类必须有一个析构函数，且只能有一个析构函数(因为析构函数无参数，不能重载，因此只能有一个)。当用户没有显示声明时，系统默认创建一个函数体为空的析构函数。
- b. 常见的构造函数和析构函数的配合方式为：在构造函数中使用 new 为对象分配空间，在析构函数中使用 delete 将分配的空间释放。

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

class Time
{
private:
    int *hour, *minute, *second;
public:
    Time(): hour(new int(10)), minute(new int(10)), second(new int(10)) {}
    ~Time();
    void display() {cout << *hour << ":" << *minute << ":" << *second << endl;}
};

Time::~~Time()
{
    delete hour;
    delete minute;
    delete second;
    cout << "析构函数被调用" << endl;
}

int main()
{
    Time t;
    t.display();
    return 0;
}
```

```
10:10:10
析构函数被调用

-----

Process exited after 0.2758 seconds with return value 0
请按任意键继续. . .
```

上面图片中的示例使用 new int(10) 对对象数据进行初始化，为了简洁，采用了**参数初始化表**的方式，使用 delete 进行析构。

另外，通常来讲，调用析构函数的次序与调用构造函数的次序相反。例如创建若干个对象 Time t1, t2, t3；由于构造函数在创建对象后立即调用，因此构造函数的调用顺序为 t1, t2, t3，而析构函数的调用顺序为 t3, t2, t1，最后执行 return 0。

详细的析构函数调用方式如下：

- 定义的全局对象(在所有函数以外)，它的构造函数会在所有函数执行之前被调用，它的析构函数在退出程序(return 0)之前，最后一个被调用。
- 定义的局部自动对象(函数体 f 中定义的对象)，它的构造函数在每次调用函数 f 时都会被调用，它的析构函数在退出 f 时被调用。
- 定义的局部静态对象(函数体 f 中定义的 static 对象，如 static Time t;)，它的构造函数仅仅在第一次调用 f 时被调用，退出 f 时不会调用析构函数。在 return 0 被调用之前，static 对象的析构函数会被调用。
- 在 return 0 被调用之前如果有多个对象的析构函数需要被调用，那么调用次序与它们的构造函数的调用次序相反。

17. 对象数组和对象指针：

C++支持**对象数组**

声明方式为 **类名 数组名[数组长度]**；

访问方式为 **数组名[下标]. 成员名**；

当没有显示调用构造函数，并且已经定义了默认构造函数时，系统将会自动将缺少的对象使用默认构造函数进行初始化。

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

class Person
{
private:
    string name;
    int age;
public:
    Person(string p_name="Default", int p_age=0): name(p_name), age(p_age) {}
    ~Person(){cout << "析构函数被调用" << endl;}
    void display(){cout << "Name : " << name << "\nAge : " << age << endl;}
};

int main()
{
    Person p[3] = {Person("Amy", 10), Person("Daming", 20)};
    for (int i=0; i<3; i++)
        p[i].display();
    return 0;
}
```

```
Name : Amy
Age : 10
Name : Daming
Age : 20
Name : Default
Age : 0
析构函数被调用
析构函数被调用
析构函数被调用
```

从上面的示例中可以看出，我们仅仅对数组的前两个成员进行了显示定义，因此，系统自动的将最后一个成员以默认初始化的方式进行补充，使得数组成员为满，不会留有空位置。

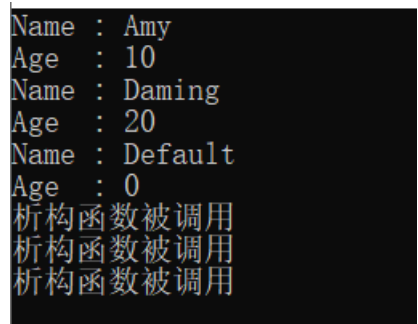
对象指针:

注意, 指针对象加减 1 的操作所对应的偏移量取决于其类型。指向对象数组的指针, 其加减操作的单位将会是一个对象。

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

class Person
{
private:
    string name;
    int age;
public:
    Person(string p_name="Default", int p_age=0): name(p_name), age(p_age) {}
    ~Person(){cout << "析构函数被调用" << endl;}
    void display(){cout << "Name : " << name << "\nAge : " << age << endl;}
};

int main()
{
    Person P[3] = {Person("Amy", 10), Person("Daming", 20)}, *p;
    p = P;
    for (int i=0; i<3; i++)
    {
        p->display();
        p += 1;
    }
    return 0;
}
```



```
Name : Amy
Age : 10
Name : Daming
Age : 20
Name : Default
Age : 0
析构函数被调用
析构函数被调用
析构函数被调用
```

注意, 数组名即为数组的首地址, 因此上面示例中使用的是 `p=P` 而不是 `p=&P`。指针加减操作的基本单位由指针的类型决定。

指向类成员的指针:

C++提供了一种特殊的指向类成员的指针, 这种指针可以指向类的数据成员(public)或者是类的成员函数(public)。通过这种指针可以访问类的公有数据成员与公有成员函数。

首先看一下普通的指针：

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

class Person
{
public:
    string name;
    int age;

    Person(string p_name="default", int p_age=0): name(p_name), age(p_age) {}
    ~Person(){cout << "析构函数被调用" << endl;}
    void display(){cout << "Name : " << name << "\nAge : " << age << endl;}
};

int main()
{
    Person P("Amy", 20);
    string *p;
    p = &P.name;
    cout << *p << endl;

    return 0;
}
```

Amy
析构函数被调用

上面的示例是普通的指针，用法也与平常一致，仅指向一个对象的某个成员。下面是指向类成员的指针，这类指针的特点是可以使用.*p 以及->*p，并且，这类指针是适用于该类所有的对象的，而不仅仅是一个对象。

a. 指向数据成员的指针：

所有该类的对象都可以使用该指针指向定义的类的成员。使用方式为：

数据类型 类名:: *指针名；

指针名 = &类名:: 数据成员名；

// 注意，是类名而不是对象名，因为该指针适用于所有的对象。数据类型指的
// 是指向的数据成员的类型

声明指针后，使用方式如下：

对象名.*指针名(指向类数据成员)；

指针(指向对象)->*指针名(指向类数据成员)；

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

class Person
{
public:
    string name;
    int age;

    Person(string p_name="default", int p_age=0): name(p_name), age(p_age) {}
    ~Person(){cout << "析构函数被调用" << endl;}
    void display(){cout << "Name : " << name << "\nAge : " << age << endl;}
};

int main()
{
    Person P1("Amy", 20), P2("Daming", 30), *p1;
    p1 = &P1;
    string Person::*p2;
    p2 = &Person::name;

    cout << p1->*p2 << "\n" << P2.*p2 << endl;
    return 0;
}
```

Amy
Daming
析构函数被调用
析构函数被调用

从上面的示例可以看出，所有对象都可以通过定义的指向 name 的指针访问自己的成员，但是，前提是，该成员为公有变量。另外，注意区别术语，上面例子中，P 为对象名而不是类名，类名为 Person，P 仅仅为类的一个实例，是对象名而不是类名。

b. 指向成员函数的指针：

同理，指向成员函数的指针可以供所有该类的对象使用，以调用自己的成员函数。声明方式与使用方式如下：

返回值类型 (类名:: *指针变量名) (参数表); // 声明指针

指针变量名 = &类名:: 成员函数名; // 初始化指针，成员函数名后无括号

(对象名.*指针变量名) (参数表); // 使用指针

(指向对象的指针->*指针变量名) (参数表); // 使用指针

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

class Person
{
public:
    string name;
    int age;

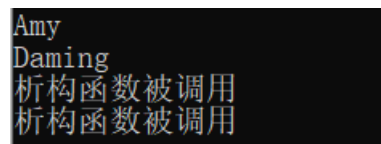
    Person(string p_name="default", int p_age=0): name(p_name), age(p_age) {}
    ~Person(){cout << "析构函数被调用" << endl;}
    void display(){cout << "Name : " << name << "\nAge : " << age << endl;}
};

int main()
{
    Person P("Amy", 20), *p1;
    p1 = &P;

    void (Person::*p2)();
    p2 = &Person::display;

    (P.*p2)();
    (p1->*p2)();

    return 0;
}
```



```
Amy
Daming
析构函数被调用
析构函数被调用
```

上述示例中仅仅创建了一个对象，如果创建多个对象，则每个对象都可以使用指针 p2 来调用 display 函数。

由于类在实例化为对象时，每个对象拥有自己的成员数据，但是，**成员函数是所有的对象所共享的**，在调用时依赖于 this 指针进行辨别。由于成员函数的共享，因此可以使用上述的指向类成员(数据成员与成员函数)的指针。

18. 静态成员：实现多个对象之间的数据共享

全局变量可以实现多个对象之间的数据和函数共享，但是全局变量的安全性比

较差，因此，通过定义静态数据成员的方式来实现对象间的共享是常用的方式。

在声明类之后，可以通过类创建对象，这时，创建的每个对象都有一份类数据成员的拷贝，而类成员函数则是所有对象共享的。与类成员函数相似，类中定义的静态对象也仅仅只有一份，由所有从属于该类的对象维护和使用，这便是使用静态数据成员实现共享的原因。

静态数据成员由 `static` 修饰，访问权限同样限制为 `public/protected/private`，静态数据成员的初始化非常特殊。

静态成员定义后必须初始化。由于静态数据成员属于类，而不属于对象，因此不能在构造函数(由对象调用)中初始化，但然，在类的声明中也不能进行初始化操作，这是因为类为抽象的，不占据存储空间。因此，类的静态成员只能在类外进行初始化操作，语法为 **数据类型 类名:: 静态数据成员名 = 初始值**；注意，初始化时不需要标记 `static` 修饰，因为已经声明过，另外，如果不进行显式的初始化操作，那么**系统将自动的把静态成员初始化为 0 (有的编译器要求显式初始化)**。

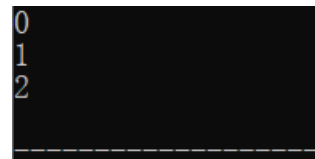
由于静态成员为类所有，因此可以使用 **类名:: 静态成员名** 进行访问，淡然也可以使用 **对象. 静态成员名** 或者 **(指向对象的指针)->静态成员名** 进行访问。

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

class Increment
{
public:
    static int begin;
    void plus(){begin += 1;}
    void display(){cout << begin << endl;}
};

int Increment::begin = 0;

int main()
{
    Increment n1, n2, n3;
    n1.display();
    n1.plus();
    cout << Increment::begin << endl;
    n2.plus();
    cout << n3.begin << endl;
    return 0;
}
```



```
0
1
2
```

上面的示例很清晰的展示了静态成员的工作方式以及不同的引用方法。另外，类的成员函数也可以定义为静态的。静态成员函数只能操纵类的静态数据成员，其他非静态数据成员不能操作。另外，静态数据成员与静态成员函数都为类所有，这意味着，即使没有创建对象，我们也可以使用 **类名:: 静态数据成员名** 或者 **类名:: 静态成员函数名** 进行调用。

另外，静态成员函数中没有 `this` 指针，这意味着如果要访问类的成员，必须指明对象名，也就是说，要想在静态成员函数中访问非静态数据成员，必须传入一个对象，利用 **对象名. 数据成员名** 进行访问。

非静态成员函数既可以访问静态数据成员，也可以访问非静态数据成员。静态成员函数可以访问静态数据成员，由于没有 `this` 指针，**访问非静态数据成员时需要借助于明确的对象名**。示例如下：

```

#include<iostream>
using namespace std;

class Complex
{
private:
    static double real;
    double imag;
public:
    Complex(){imag = 0;}
    static void display(Complex &c){cout << real << "+" << c.imag << "i" << endl;}
    ~Complex(){};
};

double Complex::real = 0;

int main()
{
    Complex c;
    c.display(c);
    return 0;
}

```

0+0i

可以看出，静态数据成员 `real` 可以被静态成员函数直接调用，而非静态成员函数 `imag` 需要指定对象名才可以被调用，这是因为静态成员函数没有 `this` 指针，无法确定是哪一个对象在调用自己。

19. 类的组合 (composition):

类的组合指的是将一个类的对象作为另一个类的数据成员。这种情况是有必要的，例如，我们定义了 `Person` 类与 `Group` 类，而 `Person` 是组成 `Group` 的基本单位，因此，将实例化后的 `Person` 对象作为 `Group` 的数据成员是合适的。`Person` 称为 `Group` 的子对象或对象成员。

```

#include<iostream>

using namespace std;

class Birthday
{
private:
    int year, month, day;
public:
    Birthday(int y, int m, int d){year = y, month = m, day = d;}
    void display();
};

void Birthday::display()
{
    cout << "Birthday : " << year << " / " << month << " / " << day << endl;
}

class Person
{
private:
    string name;
    Birthday birthday;
public:
    Person(string p_name, int y, int m, int d): name(p_name), birthday(y, m, d){}
    void display();
};

void Person::display()
{
    cout << "Name : " << name << endl;
    birthday.display();
}

```

```
int main()
{
    Person P("Amy", 1998, 11, 23);
    P.display();
    return 0;
}
```

```
Name : Amy
Birthday : 1998 / 11 / 23
```

需要注意的一点是，在 Person 中对子对象(birthday)初始化时，**必须使用参数初始化表**的方式。当含有多个子对象时，它们的构造函数被调用的顺序与它们被声明的顺序一致，与参数初始化表无关。

子对象的构造函数是通过子对象的对象名进行调用的，如示例中的 birthday(y, m, d) 当创建对象的，先按照声明顺序依次调用子对象的构造函数，最后调用本类别的构造函数。析构函数的调用顺序刚好相反。

20. 友元:

类的主要特点之一是数据隐藏，类的保护成员与私有成员在类外是无法访问的。友元可以解决这一问题。友元的目的是使得类的私有成员或保护成员可以被某些特定的函数或其它类访问。

友元往往分为两大类。如果友元是普通函数或者其它类的成员函数，则被称为友元函数；如果友元是一个类，则被称为友元类。注意，友元类中的所有成员函数都是友元函数。

友元函数不是当前类的成员函数，但它可以访问当前类的所有数据成员，包括保护成员与私有成员。

由于友元函数不存在公有或私有的界限(因为是外部函数)，因此，声明友元函数时，既可以在 public 中声明，也可以在 private 中声明，并无区别。另外，通常将友元函数定义在类的外部，毕竟友元函数不是该类的成员函数。

友元函数:

```
#include<iostream>
#include<math.h>
using namespace std;

class Point
{
private:
    double x, y;
public:
    Point(double a=0, double b=0): x(a), y(b) {}
    friend void dist(Point &p1, Point &p2);
};

void dist(Point &p1, Point &p2)
{
    double d;
    d = sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
    cout << "P1与P2的欧式距离为 : " << d << endl;
}

int main()
{
    Point p1(0, 1), p2(1, 1);
    dist(p1, p2);
    return 0;
}
```

```
P1与P2的欧式距离为 : 1
```


上面的示例中，dist 函数在被声明为友元后，可以访问 Point 类的 private 成员数据。另外，类实例化后的对象名即为对象的首地址，因此，传参时不需要再次使用&符号。

另外，友元函数不是类的成员函数，所以在定义时不要加类作用域。并且，还是因为友元函数非类成员函数，因此不能直接引用类成员数据，需要传入对象或对象指针。当一个友元函数需要访问多个类时，应当在每个类中都声明该友元。

友元成员：

友元成员是一种特殊的友元函数。如果一个类的成员函数是另一个类的友元函数，那么这个函数被称为友元成员。友元成员的特殊之处在于，它可以访问自己类的所有成员，也可以访问被声明为友元的类的所有成员。

当一个类的成员函数为另一个类的友元函数时，必须先定义该成员函数所在的类，在定义这个类之前，对另一个类进行声明。如下面的示例中，Time 类中的 display 函数为 Date 类的友元函数，因此必须先定义 Time 类，在定义 Time 类之前，必须对 Date 类进行声明：

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

class Time
{
private:
    int hour, minute, second;
public:
    Time(int h, int m, int s): hour(h), minute(m), second(s){}
    void display(Date &d);
};

class Date
{
private:
    int year, month, day;
public:
    Date(int y, int m, int d): year(y), month(m), day(d){}
    friend void Time::display(Date &d);
};

void Time::display(Date &d)
{
    cout << d.year << "-" << d.month << "-" << d.day << " ";
    cout << hour << ":" << minute << ":" << second << endl;
}

int main()
{
    Date d(2019, 12, 30);
    Time t(14, 39, 34);
    t.display(d);
    return 0;
}
```

2019-12-30 14:39:34

友元类：

当一个类为另一个类的友元时，称这个类为友元类，这个类的所有成员函数都是另一个类的友元函数。声明方式为：

friend class 友元类名； 或者是 **friend 友元类名；**

注意以下几点：

- 友元关系是单向的。如果 B 是 A 的友元类，那么 B 的成员函数可以访问 A 的数据成员，但是 A 的成员函数不能访问 B 的非公数据成员。
- 友元关系不能传递。类 C 是类 B 的友元类，类 B 是类 A 的友元类，若不加显式声明，类 C 与类 A 之间不存在友元关系。

c. 友元的引入破坏了类对信息的隐蔽性，应慎重使用。

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

class Teacher
{
private:
    int answer;
public:
    Teacher(int a=10): answer(a){}
    void check(Student &stu);
};

class Student
{
private:
    int my_answer;
public:
    friend Teacher;
    Student(int m=20): my_answer(m){}
};

void Teacher::check(Student &stu)
{
    if (answer == stu.my_answer)
        cout << "Right" << endl;
    else
        cout << "Wrong" << endl;
}

int main()
{
    Teacher t;
    Student stu;
    t.check(stu);
    return 0;
}
```



Wrong

在上面的示例中，Teacher 为 Student 的友元类，Teacher 的成员函数 check 可以查看 Student 的私有变量 my_answer。在定义时，还是先定义友元类，并且在友元类定义之前，用 class Student; 对 Student 进行声明。

21. 运算符重载

运算符重载是重载的重要表现方式。在 C++ 中，运算符重载无处不在。例如简单的加法 “+” 运算符，它可以实现 int、float、double 等等类型的加法运算就是依赖运算符重载完成的。

运算符是一种特殊的函数，运算符重载是通过运算符函数的重载实现的。重载的声明如下：

返回值类型 operator 运算符(形参表) { // 函数体; }

通常来讲，运算符是由左运算数调用，右运算数作为参数传入。运算符是左运算数类的一个成员函数，因此可以直接访问左运算数的所有属性(attribute)，所以，运算符函数往往只需要右运算数作为参数。

运算符重载应当遵循的规则：

a. 不是所有的运算符都可以重载。“? :”，“.”，“*”，“::”，“sizeof” 都不可

以重载。它们分别代表三目运算符，成员操作，指针操作，作用域限定以及求字节数。

- b. 运算符重载不改变运算符的优先级和结合性，重载后，功能与原功能往往相似。
- c. 重载运算符不改变运算符的操作个数，即原来为双目运算符，则重载后，应当仍为双目运算符。
- d. 重载的运算符要求至少有一个操作数为自己定义的类的对象。特别的，对于双目运算符，左操作数一定是自己定义的类的对象。
- e. 运算符重载函数可以是类的成员函数，类的友元函数，或者是普通函数。

重载双目运算符：

双目运算符的重载要求左操作数必须为自己定义的类的对象。对右操作数无要求。例如之前介绍过的对复数类 Complex 重载 “+” 运算符：

a. 利用成员函数进行重载：

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

class Complex
{
private:
    double real, imag;
public:
    Complex(double r=0, double i=0): real(r), imag(i){}
    Complex operator +(Complex &c);
    void display();
};

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

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

int main()
{
    Complex a(1, 1), b(1, 1), c(a + b);
    c.display();
    return 0;
}
```



2+2i

在上面的示例中，使用成员函数对 “+” 运算符进行了重载，重载后的运算符函数是 Complex 类的成员函数。对象 c 的初始化方式为**对象的复制**。

b. 利用友元函数进行重载：

友元函数不是类的成员函数。因为被类声明为友元的函数可以访问类的全部数据成员，因此，可以实现运算符重载。

但是友元函数访问类的数据成员时，**必须借助于对象名或者对象指针**，因此，对于双目运算符来说，需要传入两个对象。例如，用类的成员函数对 “+” 进行重载，利用友元函数对 “-” 进行重载。示例如下：

```

#include<iostream>
using namespace std;

class Complex
{
private:
    double real, imag;
public:
    Complex(double r=0, double i=0): real(r), imag(i){}
    Complex operator +(Complex &c){return Complex(real + c.real, imag + c.imag);}
    friend Complex operator -(Complex &c1, Complex &c2);
    void display(){cout << real << "+" << imag << "i" << endl;}
};

Complex operator -(Complex &c1, Complex &c2)
{
    return Complex(c1.real - c2.real, c1.imag - c2.imag);
}

int main()
{
    Complex a(1, 1), b(1, 1), c(a + b), d(a - b);
    c.display();
    d.display();
    return 0;
}

```

2+2i
0+0i

注意，重载“-”运算符前面的 Complex 是返回值类型，不是指定作用域。重载加法“+”时，第一个 Complex 是返回值类型，第二个 Complex:: 是作用域，表明这个函数是 Complex 类的成员函数。

重载单目运算符：

由于单目运算符只有一个操作数，因此这个操作数必须是对象。常见的单目运算符为“++”操作与“--”操作。并且，这两个运算符十分特殊，因为它们位于运算对象的前后所发挥的作用不一样。k++将会执行对象 k 的自增操作，但是返回的是 k 原来的数值；++k 也会执行对象 k 的自增操作，并且返回自增后的数值。“--”操作也是一样的。

为了区别前置++与后置++，C++规定，后置自增运算的参数表中添加一个 int 形参，实际使用时无需传入。

同理，单目运算符的重载可以通过类的成员函数实现，也可以通过友元函数实现，实质上只要可以访问到对象的数据成员即可，如果数据成员都是 public 的，那么普通的函数也可以实现重载。

```

#include<iostream>
using namespace std;

class Int
{
private:
    int n;
public:
    Int(int m=0): n(m){}
    Int operator ++();
    friend Int operator ++(Int &N, int);
    void display(){cout << n << endl;}
};

Int Int::operator ++()
{
    n++;
    return *this;
}

Int operator ++(Int &N, int)
{
    Int temp(N);
    N.n++;
    return temp;
}

int main()
{
    Int n;
    n.display();
    (n++).display();
    n.display();
    (++n).display();
    return 0;
}

```

0
0
1
2

由于友元函数无法直接访问类成员变量，因此，我们必须传入对象地址或者是指向对象的指针。而使用类成员函数进行运算符重载时就不需要传入对象信息。“--”操作的重载方式和“++”一样，**用 int 形参来表示后置操作，不加该形参时代表前置操作。**

另外，上面的*this 代表当前对象，后置++操作的重载中，使用了对象复制。友元函数中无法使用 this 指针，因此必须传入对象地址或指向对象的指针。

特殊运算符的重载：

常见的复合运算符以及判断运算符包括：+=, -=, <, >, ==, <=, >=, !=等等，这些运算符的重载方式和之前介绍的没有什么不同，例如在 Complex 类中，+=就需要分别对实部和虚部进行加法操作，然后返回一个 Complex 类的对象。

下标运算符“[]”的重载比较特殊。因为我们调用 a[i]时希望能得到它的值，并且又能够对 a[i]进行赋值，因此，我们需要返回一个变量引用，而不是变量值本身。变量引用为：int& float& double& 等等。

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

class array
{
private:
    int a[5];
public:
    array(){};
    int& operator [](int i);
    void display();
};

int& array::operator [](int i)
{
    if (i < 5 || i > -1)
        return a[i];
}

void array::display()
{
    for (int i=0; i<5; i++)
        cout << a[i] << " ";
}
```

```
int main()
{
    array a;
    for (int i=0; i<10; i++)
        a[i] = i * i;
    a.display();
    return 0;
}
```

0 1 4 9 16

Process exited after 0.17 seconds v
请按任意键继续. . .

从上面的示例可以看出，返回变量的引用可以实现赋值操作，也可以实现变量值的查看。另外，在“[]”的重载中，我们限制了长度，因此，虽然在主函数中下标范围超出了数组，并不会因此产生非法赋值操作。

另外，流运算符“<<”与“>>”也是可以重载的，返回类型为 ostream&，参数表的第一个对象为 ostream&类型，第二个对象为自定义对象。并且，**流运算符只能作为友元函数或者普通函数进行重载，不能作为类的成员函数进行重载**，这是因为左运算数只能为 ostream&类型。

在 C++ 中，**对象 A 运算符 对象 B** 将会被解释为 **对象 A. 运算符(对象 B)**；

22. 继承和派生

继承 (inheritance) 机制是面向对象技术提供了一种用于解决软件复用

(software reuse)问题的途径。继承是指，再定义一个新的类时，先把一个或多个已有类的功能全部包含进来，然后再给出新功能的定义或对已有类的某些功能进行重定义。

类的继承就是在已有类(基类，又叫做父类)的基础上创建新类的过程，创建的新类称为派生类(也叫做子类)。

派生类是基类的具体化，基类是派生类的抽象。例如，我们创建一个 Student 类，这个类所适用的范围十分广泛，然后我们在 Student 类的基础上创建了更为具体的本科生 Undergraduate 类。这里，Student 类即为基类(父类)，Undergraduate 类即为派生类(子类)。

继承的声明方式为：

class 派生类名：继承方式 基类名 { //声明派生类数据成员与成员函数}；

其中，继承方式为 public/protected/private 中的一种，继承方式指定了派生类成员以及类外对象对从基类继承来的成员的访问权限。可以不显示声明，系统默认为 private 继承。

继承时，派生类可以完成的主要功能为：

- a. 增加新的数据成员。
- b. 增加新的成员函数。
- c. 对基类成员进行重定义。
- d. 改变基类成员在派生类中的访问属性。

派生类会继承基类所有的数据成员和成员函数(不包括基类的构造函数与析构函数)，派生类应当自己定义合适的构造函数和析构函数。

在派生类中，可以定义与基类成员同名的成员，这样，派生类中的新成员将会覆盖基类中的同名成员。另外，如果要在派生类中定义成员函数来覆盖基类的成员函数，那么该成员函数的函数名以及参数个数和参数类型都必须与基类中的成员函数完全一样。

派生类继承方式：

派生类的继承方式会影响基类成员(包括数据成员和成员函数)在派生类中的访问属性。私有成员与受保护的成员之间的差异主要体现在继承中。受保护的成员是指：该成员不能被外界(类外)引用，但是可以被派生类中的成员引用。

继承方式与对应的成员限制变化如下：

基类成员属性	public	protected	private
公有(public)继承	public	protected	派生类内外均无法访问
保护(protected)继承	protected	protected	派生类内外均无法访问
私有(private)继承	private	private	派生类内外均无法访问

可以看出，基类中 private 成员无论通过什么继承方式继承，派生类都无法访问。public 继承没有改变成员的访问权限，private 继承一次后，成员仅派生类内可以访问，继承两次后，成员在派生类内也无法访问。

```

#include<iostream>
using namespace std;

class Base
{
private:
    int pri;
protected:
    int pro;
public:
    int pub;
    Base(int p1=0, int p2=0, int p3=0): pri(p1), pro(p2), pub(p3){}
};

class Child:private Base {};

```

上图中定义了一个空的子类，仅仅是为了检测不同继承方式下基类成员的访问权限。访问权限结果如下：

继承方式	类内可访问	类外可访问	内外都可访问	内外不可访问
public	pub, pro	pub	pub	pri
protected	pub, pro	无	无	pri
private	pub, pro	无	无	pri

派生类的构造函数和析构函数：

构造函数：

派生类会继承基类的所有成员，基类成员的初始化由基类构造函数完成。同时，在派生类中，我们又可以定义新的派生类成员，这部分成员的初始化由派生类的构造函数完成。然而，需要注意的是，派生类并不会继承基类的构造函数和析构函数，因此我们需要在派生类的构造函数中对基类成员初始化所需要的参数进行设置。

分为以下几个情况：

- 基类构造函数无参数或带有全部默认参数，或没有显式定义构造函数，那么派生类可以不向基类传递初始化参数，没有需求的话，甚至可以不定义构造函数。如下图示例：

```

#include<iostream>
using namespace std;

class Base
{
private:
    int pri;
protected:
    int pro;
public:
    int pub;
    Base(int p1=0, int p2=0, int p3=0): pri(p1), pro(p2), pub(p3){}
};

class Child:public Base {};

int main()
{
    Child c;
    return 0;
}

```

- 基类构造函数带有参数时，派生类必须定义构造函数，用来向基类传递参数以完成基类成员的初始化操作。C++中派生类的构造函数格式大致有两种，分别是普通构造函数格式和初始化参数列表的形式，具体如下：

派生类名(参数总表): 基类名(参数值), 派生类成员(参数值) {};

派生类名(参数总表): 基类名(参数值) { //派生类构造函数体; }

若将构造函数定义在派生类外, 则声明格式仍然为:

派生类名(参数总表);

注意, 上面两种方式中, 基类名()中是参数值, 而不是参数表, 因此不应该写变量的类型。派生类名()中是参数总表, 要把用到的所有初始化参数值以及类型都写进去。示例如下:

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

class Base
{
public:
    int base;
    Base(int p): base(p){}
};

class Child:public Base
{
public:
    int child;
public:
    Child(int p1, int p2):Base(p1), child(p2){}
    void display(){cout << base << " " << child << endl;}
};

int main()
{
    Child c(1, 2);
    c.display();
    return 0;
}
```

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

class Base
{
public:
    int base;
    Base(int p): base(p){}
};

class Child:public Base
{
public:
    int child;
public:
    Child(int p1, int p2);
    void display(){cout << base << " " << child << endl;}
};

Child::Child(int p1, int p2):Base(p1)
{
    child = p2;
}

int main()
{
    Child c(1, 2);
    c.display();
    return 0;
}
```

上图中左侧为初始化参数表的方式, 右侧为普通的初始化方式。通常采用初始化参数表, 更加简洁。并且, 在类的组合中, 类对其内的子对象的初始化必须使用初始化参数表的方式。详见 19 类的组合(composition)。

析构函数:

在派生类释放时, 析构函数的调用顺序与构造函数相反。构造函数的调用顺序为先调用基类, 后调用派生类; 析构函数的构造顺序为先调用派生类, 再调用基类。

含有子对象的派生类构造函数:

子对象的概念是指类中的某个数据成员是另一个类的某个实例化对象, 因此被称为子对象。含有子对象的派生类在初始化时会按顺序完成以下操作:

- 对基类数据成员初始化。
- 对子对象数据成员初始化。
- 对派生类数据成员初始化。

构造函数的格式为:

派生类名(参数总表): 基类名(参数值), 子对象名(参数值), 新成员(参数值) {};

派生类名(参数总表): 基类名(参数值), 子对象名(参数值) { //构造函数体; }

上面两种格式分别为初始化参数表和普通构造函数格式, 另外, 基类名和子对象名的位置是任意的, 但是构造函数的调用次序是固定的: 先基类, 再子对象, 最后是派生类。析构函数的调用次序与构造函数相反。示例如下:


```

#include<iostream>
using namespace std;

class Date
{
public:
    int year, month;
    Date(int y, int m): year(y), month(m){}
};

class Time
{
public:
    int hour, minute, second;
    Time(int h, int m, int s): hour(h), minute(m), second(s){}
};

class Calendar: private Date
{
private:
    int day;
    Time t;
public:
    void display();
    Calendar(int y, int m, int d, int h, int mi, int s): Date(y, m), t(h, mi, s), day(d){}
};

void Calendar::display()
{
    cout << "现在的时刻是 : " ;
    cout << year << "-" << month << "-" << day << " " ;
    cout << t.hour << ":" << t.minute << ":" << t.second << endl;
}

int main()
{
    Calendar now(2019, 12, 31, 16, 49, 40);
    now.display();
    return 0;
}

```

现在的时刻是 : 2019-12-31 16:49:40

在上面的示例中，Calendar 类继承了 Date 类，并且包含了子对象 t，其中 t 是类 Time 的实例化。对与对象 t 中成员的使用，需要借助于对象名。

多级派生:

之前的示例都是单级派生，也就是子类 and 父类之间仅有一次继承关系。例如，创建了基类 A，类 B 通过继承基类 A 创建，类 C 通过继承类 B 创建，那么称：A 是 B 的直接父类，A 是 C 的间接父类，B 是 C 的直接父类。A 与 B，B 与 C 之间都是单级派生关系，A 与 C 之间是多级派生关系。

在派生类 C 中，构造函数的调用顺序为：先间接父类 A，再直接父类 B，最后派生类 C。析构造函数的调用顺序相反。

另外，注意，派生类 C 仅仅调用直接父类 B 的构造函数，间接父类 A 的构造函数是由 B 调用的，而不是 C 调用的。因此，派生类 C 的构造函数仅仅列出 B 即可。格式为：

C(参数总表): B(参数值), 新成员(参数值){};

C(参数总表): B(参数值){//构造函数体; }

上面两种写法分别是初始化参数表形式和普通构造函数形式，二者均可以完成派生类以及各级基类的参数初始化任务。示例如下：

```

#include<iostream>
using namespace std;

class Year
{
public:
    int year;
    Year(int y): year(y){}
};

class Month: public Year
{
public:
    int month;
    Month(int y, int m): Year(y), month(m){}
};

class Day: public Month
{
public:
    int day;
    Day(int y, int m, int d): Month(y, m), day(d){}
    void display(){cout << year << "-" << month << "-" << day << endl;}
};

int main()
{
    Day d(2019, 12, 31);
    d.display();
    return 0;
}

```

2019-12-31

上述示例中展示了 Day 继承 Month，Month 继承 Year 的过程。Day 的构造函数仅仅为 Month 初始化，Year 的初始化由传入 Month 的参数以及 Month 的构造函数完成调用。

另外，如果基类为带参构造函数，但是派生类又不需要为新增的成员初始化时，可以将派生类的构造函数写成空函数：

派生类名(参数表)：基类名(参数值) {}

若基类为默认参数构造函数或者无参构造函数或者为系统默认构造函数时，派生类可以不显式调用基类构造函数，而直接写成：

派生类名(参数表)：派生类成员(参数值) {}

最后，若派生类与基类都是系统默认构造函数，或者派生类为系统默认，基类为非显式调用时，可以直接省略派生类的构造函数。

关于析构函数，派生类不会继承基类的析构函数。在执行派生类的析构函数时，系统会自动调用基类以及子对象的析构函数，调用顺序与构造函数相反。

多重继承：派生类继承了多个基类

多重继承仅仅是派生类继承了多个基类而已，派生类与每个基类的关系同单继承完全一致。从而，多重继承实质上是多个单继承的合并。

多重继承时，派生类的声明方式为：

```

class 派生类名: 继承方式1 基类名1, 继承方式2 基类名2
{
    //派生类函数体;
};

```

上面的基类可以按照格式继续增加，并无数量限制。另外，多重继承下，派生类的构造函数与单继承相似，仍然要包含参数总表，普通构造方式与参数初始化表都可以。

多重继承中，在派生类实例化为对象时，先调用基类的构造函数，再调用派生类的构造函数，若干个基类构造函数的调用顺序取决于派生类在声明时的继承顺序，

析构函数的调用顺序与构造函数相反。构造函数格式为：

派生类名(参数总表):基类名 1(参数值),基类名 2(参数值),新成员(参数值) {};

派生类名(参数总表):基类名 1(参数值),基类名 2(参数值) { //新成员初始化; }

如果将构造函数定义在类外,则在类内需要使用 **派生类名(参数总表);** 进行声明,在类外定义时,需要加上作用域声明 **派生类名::**。示例如下:

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

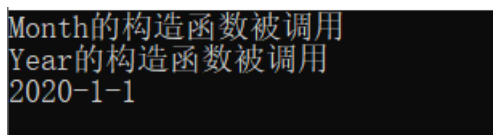
class Year
{
public:
    int year;
    Year(int y){year = y; cout << "Year的构造函数被调用" << endl;}
};

class Month
{
public:
    int month;
    Month(int m){month = m; cout << "Month的构造函数被调用" << endl;}
};

class Day: private Month, private Year
{
private:
    int day;
public:
    Day(int y, int m, int d): Year(y), Month(m), day(d){}
    void display();
};

void Day::display()
{
    cout << year << "-" << month << "-" << day << endl;
};

int main()
{
    Day d(2020, 01, 01);
    d.display();
    return 0;
}
```



```
Month的构造函数被调用
Year的构造函数被调用
2020-1-1
```

从上面的示例可以看出,多重继承构造函数的调用顺序与派生类中赋值顺序无关,而取决于派生类在声明继承时,基类的声明顺序。另外,上述示例中使用 01 时有风险的。以 0 开头的数字为 8 进制,以 0x 开头的数字为 16 进制,但是无论在什么进制下,1 都代表 10 进制下的数字 1,但是 010 就是 10 进制下的 8,0x10 就是 16 进制下的 16,应当注意这一点。

另外,析构函数的调用顺序与构造函数相反,先派生类析构,再基类析构,基类析构的顺序与基类构造的顺序相反。

最后,如果派生类为多重继承(注意区分**多重继承**与**多级派生**),并且派生类中也包含有子对象,那么构造函数的调用顺序为:先基类(按照声明的继承顺序),再子对象,最后为派生类。析构函数的调用顺序与此相反。

限定访问与同名覆盖:

限定访问与同名覆盖主要存在于多重继承的情况中。如果派生类继承的多个基类中有同名成员,无论是同名数据成员还是成员函数,则派生类在访问这些成员时

可能会产生冲突。此时，应当限定成员的访问域以避免冲突。

例如类 C 多重继承类 A 和类 B，类 A 与类 B 中都有一个 int 型成员变量 x，则在类 C 内使用 x 访问（或者类 C 外，使用 c.x）会报错，产生错误的原因是因为访问不明确。解决方法为成员名限定（假设 c 为类 C 的一个实例化对象）：

在类 C 内访问：A::x；或者 B::x；

在类 C 外访问：c.A::x；或者 c.B::x；

也就是说，在访问的时候要显式的指定访问对象所在的基类。示例如下：

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

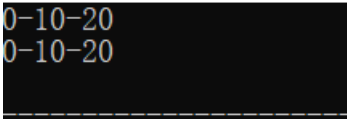
class A
{
public:
    int x;
    A(int a): x(a){}
};

class B
{
public:
    int x;
    B(int b): x(b){}
};

class C: public A, public B
{
public:
    int x;
    C(int a=0, int b=10, int c=20): A(a), B(b), x(c){}
    void display();
};

void C::display()
{
    cout << A::x << "-" << B::x << "-" << C::x << endl;
}

int main()
{
    C c;
    c.display();
    cout << c.A::x << "-" << c.B::x << "-" << c.C::x << endl;
    return 0;
}
```



```
0-10-20
0-10-20
```

上面的示例展示了类内和类外如何对同名成员进行访问，类内直接使用作用域运算符即可，类外需要使用 **对象名.类名::成员名**。

同名覆盖：

上面的示例中，在派生类中也定义了一个名为 x 的 int 型变量，在访问时，我们使用的是 C::x 或者 c.C::x，其实，这是完全没有必要的。

当派生类中新定义了与基类中同名的数据成员或者成员函数时，如果我们不加限定的访问，即直接访问 x 或者 c.x，是会产生歧义的。因为基类成员会被派生类成员**同名覆盖**。因此，在不加作用域限定访问时，系统会默认访问派生类中新定义的成员。

值得说明的一点是，同名覆盖并不会清除基类中的 x 成员，这从上面的示例可以看出。总之，**在同名覆盖后，若不加限定的访问，那么会默认访问派生类成员；如果想要访问基类成员，则需要加作用域限定。同名覆盖后，基类同名成员依然存在。**

另外，成员函数的同名覆盖要求函数名与参数个数以及参数类型完全相同，如果仅仅函数名相同，而参数列表存在差异，那就不是同名覆盖，而是函数重载。

```

#include<iostream>
using namespace std;

class A
{
public:
    int x;
    A(int a): x(a){}
};

class B
{
public:
    int x;
    B(int b): x(b){}
};

class C: public A, public B
{
public:
    int x;
    C(int a=0, int b=10, int c=20): A(a), B(b), x(c){}
    void display();
};

void C::display()
{
    cout << A::x << "-" << B::x << "-" << C::x << endl;
}

int main()
{
    C c;
    c.display();
    cout << c.A::x << "-" << c.B::x << "-" << c.x << endl;
    return 0;
}

```

该示例仅仅去除了派生类作用域限定，运行结果与上个示例完全一致。

赋值兼容:

由于派生类包含了全部的基类对象，因此，可以使用基类对象的地方，使用派生类替代也是可以的，这便是兼容。总之，派生类可以替代基类，但是基类不能替代派生类，因此：

- 基类对象可以赋值给基类对象，派生类对象也可以赋值给基类对象。
- 基类指针可以指向基类对象，也可以指向派生类对象。
- 基类引用可以指向基类对象，也可以指向派生类对象。

以上操作不可以反向。

另外，需要注意，当使用基类对象指针(或引用)指向派生类时，仅能调用自己的基类成员而不能调用派生类中新定义的成员。

```

#include<iostream>
using namespace std;

class A
{
public:
    int x;
    A(int a): x(a){}
    void ShowA(){cout << "A : " << x << endl;}
};

class B
{
public:
    int x;
    B(int b): x(b){}
    void ShowB(){cout << "B : " << x << endl;}
};

class C: public A, public B
{
public:
    C(int a, int b): A(a), B(b){}
    void Show(){cout << "A-B : " << A::x << "-" << B::x << endl;}
};

int main()
{
    A a(10), *p1;
    B b(20), *p2;
    C c(30, 40);
    p1 = &c;
    p2 = &c;

    a.ShowA();
    b.ShowB();
    c.Show();

    a = c;
    b = c;
    a.ShowA();
    b.ShowB();

    p1->ShowA();
    p2->ShowB();

    return 0;
}

```

```

A : 10
B : 20
A-B : 30-40
A : 30
B : 40
A : 30
B : 40

```

注意，上面示例中，指针 p1 仅能调用类 A 中的成员，指针 p2 仅能调用类 B 中的成员，用它们调用派生类或者其它基类成员是错误的，是非法操作。

虚拟继承：

虚拟继承是为了解决多级多重继承中出现的资源浪费问题。例如有基类 A，类 B 和类 C 由基类 A 派生而来，类 D 多重继承类 B 和类 C 而产生，这个问题中，就涉及到了多级继承与多重继承的问题，资源浪费产生的原因如下：

基类 A：

类名	数据成员	成员函数
A	int data	void fun()

派生类 B：继承基类 A

类名	数据成员	成员函数
B	int data, data_b	void fun()

派生类 C：继承基类 A

类名	数据成员	成员函数
C	int data, data_c	void fun()

派生类 D：继承类 B 与类 C

类名	数据成员	成员函数
D	B::data, C::data, data_b, data_c	B::fun(), C::fun

在派生类 D 中，继承自 B 和 C 的 fun() 都是基类 A 的 fun() 的拷贝，并且 B 和 C 的 data 也都是基类 A 中的 data 的拷贝，这便造成了资源的浪费。为了使得派生类 D 中仅有一份基类 A 中成员的拷贝(同时解决成员同名问题)，可以将共同基类 A 设置为虚基类。

虚基类的作用就是：使得公共基类的成员在其间接派生类中只保留一份拷贝(只继承一次)。也就是说，在上面例子中的 D 类中，仅仅保留一个 data 和 fun()。

另外，为了实现上述目的，必须在该基类所有的直接派生类中，将该基类声明为虚拟继承。例如：

```
class A;
class B: virtual public A;
class C: virtual public A;
class D: public B, public C;
```

使用上述方式时，便会在类 D 中仅仅保留一份共同基类 A 的成员拷贝。

虚基类初始化规则：

- 以虚基类为基类的派生类(无论是直接派生还是间接派生)，必须显式的调用虚基类的构造函数(除非虚基类构造函数无参数或全部默认参数)。
- 若派生类构造函数中包含了虚基类和非虚基类时，先调用虚基类的构造函数。
- 例如，类 A 为共同基类，类 B 与类 C 继承类 A，类 D 继承类 B 与类 C，并且虚拟继承类 A，那么在类 D 的构造函数中，


```

#include<iostream>

using namespace std;

class A
{
public:
    int a;
    A(int p, string s);
};

A::A(int p, string s)
{
    a = p;
    cout << s + "调用了A的构造函数" << endl;
}

class B: virtual public A
{
public:
    B(int p): A(p, "B"){}
};

class C: virtual public A
{
public:
    C(int p): A(p, "C"){}
};

class D: public B, public C
{
public:
    D(int p): A(p, "D"), B(p), C(p){}
};

int main()
{
    D d(10);
    return 0;
}

```

D调用了A的构造函数

Process exited after 0 seconds. Please press any key to continue...

从上面的示例可以看出，直接派生类 B 与类 C 都是继承虚基类 A 而来，间接派生类 D 继承了类 B 与类 C，因此，类 D 是虚基类 A 间接派生的类，需要在构造函数中显式调用（除非是默认参数或者无参数）虚基类 A 的构造函数。其它类的构造函数调用要求与之前的单继承一致。可以看出，类 B 与类 C 都是直接继承虚基类 A 的，在 D 调用 B 与 C 的构造函数时，B 与 C 应当调用 A 的构造函数，但是这个调用被忽略了，仅仅由类 D 直接调用了 A 的构造函数，这就是**虚拟继承**的作用。

即：**只有实际构造对象（如示例中的对象 d）的类的构造函数才会引发对虚基类构造函数的调用，而其他基类在成员初始化列表中对虚基类构造函数的调用都会被忽略，这样就可以保证派生类对象中虚基类成员只会被初始化一次，只有一份拷贝。**

23. 多态性与虚函数

多态性 (polymorphism) 指的是发出的消息被不同的对象接受时会产生完全不同的行为，表现为同一个函数名具有多种不同实现。多态的目的是为了实现**接口重用**，**函数重载（包括运算符重载）就是多态性的体现。**

结合类的概念，多态性可以理解为，由继承而产生的不同的类，在实例化为对象后，该对象对同一消息（函数调用）会做出不同的响应（调用结果）。

多态性分为两类：**静态多态性**和**动态多态性**。

静态多态性又叫做**编译时的多态性**，通过**函数重载**实现，在编译阶段系统就已经确定调用哪个函数。

动态多态性又叫做运行时的多态性，通过**虚函数** (virtual function) 实现，在程序运行过程中才动态的确定调用哪个函数。

上面所说的确定调用哪个函数的过程称为**关联** (binding)，又叫做**绑定**。关联是指当调用的函数存在若干同名函数时，如何从中选择。当然，关联也是有两类的，与多态的类别相对应，分别为**静态关联（编译链接阶段完成绑定）**和**动态关联（程序运行阶段完成绑定）**，静态关联对应于静态多态性，动态关联对应于动态多态性。

静态绑定的问题如下：

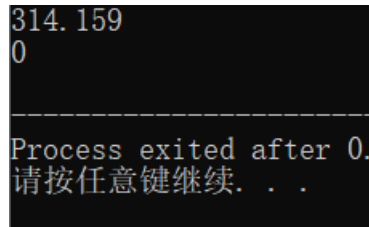
```
#include<iostream>
const double Pi = 3.1415926;
using namespace std;

class Area
{
public:
    double area(){return 0;}
    void display(){cout << area() << endl;}
};

class Circle: public Area
{
private:
    double R;
public:
    Circle(double r): R(r){}
    double area(){return Pi * R * R;}
    void display(){cout << area() << endl;}
};

void ShowArea(Area &p)
{
    cout << p.area() << endl;
}

int main()
{
    Circle c(10);
    c.display();
    ShowArea(c);
    return 0;
}
```



```
314.159
0
-----
Process exited after 0.
请按任意键继续. . .
```

在上面的示例中，兼容赋值规则使得我们可以将基类对象指针指向派生类，因此，我们可以将派生类对象 c 作为参数传给 ShowArea。另外，由于派生类中对 area() 进行了同名覆盖，因此当代用派生类的 area() 函数时，不需要指明类作用域，会默认调用派生类的成员函数。

上面程序暴露出的问题是，在编译时，ShowArea 函数已经将调用的 area() 函数指向了基类 Area，这时，即使我们传入了派生类对象 c，调用的 area 函数仍然是基类 Area 的，而不是派生类 Circle 的，这便是静态绑定的缺点，不够灵活。

但是，静态绑定所带来的好处是：由于调用关系在编译链接阶段已经完全确定，所以程序的执行效率很高。当然，这也正是动态绑定的缺点，由于在程序运行阶段确定函数调用（消息）与程序代码（方法）之间的匹配关系，导致函数调用速度慢，但是增加了编程的灵活性，问题的抽象性和程序得易维护性。

虚函数：

虚函数允许函数调用与函数体之间的关联在运行时才建立。**虚函数是基类的成员函数，在派生类中进行重载（实质是同名覆盖）**。在基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义。重新定义时，此函数的函数名，参数个数，参数类型以及参数位置都必须与基类中的完全一致，实质上就是同名覆盖。

在基类中，声明方式如下：

virtual 返回值类型 函数名(参数表) { //函数体; }

在派生类中重新定义时，不需要写 virtual 关键字（**推荐写上**），可直接按以下方式写：

返回值类型 函数名(参数表) { //函数体; }

若基类 A 中定义了虚函数 f，其派生类 B 中也定了函数 f，那么派生类中的成员函数在满足以下条件时也是虚函数，无论是否在派生类中使用了 virtual 关键字声明：

- 该函数与基类函数具有相同的函数名称。
- 该函数与基类函数具有完全相同的参数个数以及对应的参数类型。
- 该函数与基类函数具有相同的返回类型。

从而,若 f 在基类 A 中是虚函数,在派生类 B 中对 f 的定义满足上述三条,那么无论在派生类中是否添加了 `virtual` 关键字, f 都是 B 的虚函数。若不满足上述三条,则 f 不是 B 的虚函数(此时为函数重载,静态绑定),除非再添加 `virtual` 关键字使得 f 称为 B 的虚函数。

虚函数的效果:

当通过基类指针调用虚函数时,基类指针指向哪个类,就调用哪个类的对象。例如,基类 A ,成员函数为 f ,派生类 B ,重定义了成员函数 f ,定义了 $A *p, B b, p=&b$,此时,如果使用 $p->f()$ 将会调用基类 A 的成员函数 f ,而不是派生类 B 的。

如果在基类 A 中将 f 声明为虚函数,在派生类中进行重定义,再使用 $p->f()$ 将会调用派生类 B 的成员函数 $f()$ 。示例如下:

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

class A
{
public:
    void f(){cout << "A的成员函数被调用" << endl;}
};

class B
{
public:
    virtual void f(){cout << "B的成员函数被调用" << endl;}
};

class C: public A, public B
{
public:
    void f(){cout << "C的成员函数被调用" << endl;}
};

int main()
{
    A *a;
    B *b;
    C c;
    a = &c;
    b = &c;
    a->f();
    b->f();
    return 0;
}
```

```
A的成员函数被调用
C的成员函数被调用

Process exited after 0.1
请按任意键继续. . .
```

在上面的示例中,指针 b 指向的对象是 c ,并且,由于类 B 中声明了虚函数,根据 3 条规则,也可以看出,类 C 中的 f 也是虚函数。从而, $b->f()$ 调用的是派生类的函数,而 $a->f()$ 调用的是基类 A 的函数。这便是虚函数的作用。如果类 C 中的 f 不是虚函数,

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

class A
{
public:
    void f(){cout << "A的成员函数被调用" << endl;}
};

class B
{
public:
    virtual void f(){cout << "B的成员函数被调用" << endl;}
};

class C: public A, public B
{
public:
    void f(int){cout << "C的成员函数被调用" << endl;}
};

int main()
{
    A *a;
    B *b;
    C c;
    a = &c;
    b = &c;
    a->f();
    b->f();
    return 0;
}
```

```
A的成员函数被调用
B的成员函数被调用

Process exited after 0.1653
请按任意键继续. . .
```

那么调用结果如下，b->f()调用基类B的成员函数而不是C的：

最后，由于默认参数是静态绑定的，虚函数是动态绑定的，因此，当虚函数中存在默认参数时十分容易出错，很少重新定义带有默认参数的虚函数。

虚函数实现多态性的方法：

在基类A中定义虚函数f，在派生类中重新定义虚函数f，例如派生类有类B与类C。则虚函数实现多态性是依赖基类指针的，A *p，将p指向哪个类对象，p->f()就调用哪个类的成员函数f。从而，**多态是指：向不同的对象发送同一个消息(调用同样的函数)，不同的对象在接收时会产生不同的行为。**

注意，消息就是指函数调用。

另外，虚函数是类的成员函数，**不能是友元函数，不能是静态成员函数。**

在多重继承中，例如基类A与基类B分别由虚函数a()与b()，派生类C继承类A与类B后重定义虚函数a()与b()，这样，如果定义A *p，C c，p=&c，p->a()就可以调用派生类C的成员函数a()，同理，用同样的方法也可以调用C的成员函数b()。

但是，如果基类A与基类B中的虚函数一样(函数名和参数表)该怎么办？可以使用中间类来解决这个问题。

类A与类B中都有虚函数f()，为了区分，在类A_M中，重新定义f()，并引入新的空虚函数f_a()，在f()中调用f_a()。在类B中的中间类B_M中，用同样的方法定义f_b()，并在重新定义的f()中调用f_b()。

这样，就得到两个新的中间类，它们所包含的虚函数为f_a()和f_b()(f()也是虚函数，但不需要考虑)，然后类C继承A_M与B_M，并重写f_a()与f_b()，这样，当使用基类指针指向类对象c时，p->f()便会调用中间类中的f()，进而调用C中的f_a()或f_b()。示例如下：

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

class A
{
public:
    virtual void f(){cout << "A类成员函数被调用" << endl;}
};

class B
{
public:
    virtual void f(){cout << "B类成员函数被调用" << endl;}
};

class A_M: public A
{
public:
    virtual void f_a(){}
    void f(){f_a();}
};

class B_M: public B
{
public:
    virtual void f_b(){}
    void f(){f_b();}
};

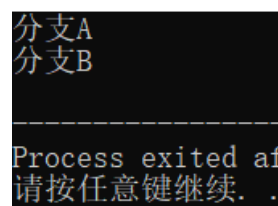
class C: public A_M, public B_M
{
public:
    void f_a(){cout << "分支A" << endl;}
    void f_b(){cout << "分支B" << endl;}
};
```

```
int main()
{
    A *a;
    B *b;
    C c;

    a = &c;
    b = &c;

    a->f();
    b->f();

    return 0;
}
```



```
分支A
分支B
-----
Process exited af
请按任意键继续.
```

这样，就可以实现对同名基类虚函数的处理。很少会遇到此类情况。

虚析构函数：

注意，**构造函数不可以声明为虚函数**，这是非法的，但是析构函数可以声明为虚函数。 如果一个类的析构函数是虚函数，那么由这个类派生而来的派生类析构函数都是虚函数，无论是否用 virtual 关键字显式声明。

之所以出现虚析构函数是因为下面的这种情况：

通常如果派生类撤销时，会先调用派生类的析构函数，然后调用基类的析构函数，释放掉空间。但是，如果用基类对象指针指向派生类对象时，在退出程序之前仅仅会调用基类析构函数，而不会调用派生类的析构函数。虚析构函数就是为了解决这个问题。示例如下：

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

class A
{
public:
    ~A(){cout << "A的析构函数被调用" << endl;}
};

class B: public A
{
public:
    ~B(){cout << "B的析构函数被调用" << endl;}
};

int main()
{
    B b;
    return 0;
}
```

B的析构函数被调用
A的析构函数被调用

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

class A
{
public:
    ~A(){cout << "A的析构函数被调用" << endl;}
};

class B: public A
{
public:
    ~B(){cout << "B的析构函数被调用" << endl;}
};

int main()
{
    A *p;
    p = new B;
    delete p;
    return 0;
}
```

A的析构函数被调用

引入虚析构函数后，是为了解决上图中右侧示例的问题，希望在 delete 基类对象指针指向的对象时，要将派生类的析构函数也调用，释放掉所有的内存占用。虚析构函数的使用方法很简单，只需要在基类和派生类的析构函数前加上 virtual 关键字即可，通常，无论是否需要显式定义析构函数，都要将析构函数定义为 virtual 的类型，一般是 virtual 的空析构函数。这样可以保证完全撤销掉动态分配的内存。示例如下(析构调用顺序为先派生类，后基类)：

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

class A
{
public:
    virtual ~A(){cout << "A的析构函数被调用" << endl;}
};

class B: public A
{
public:
    virtual ~B(){cout << "B的析构函数被调用" << endl;}
};

int main()
{
    A *p;
    p = new B;
    delete p;
    return 0;
}
```

B的析构函数被调用
A的析构函数被调用

纯虚函数与抽象类:

如果基类中不易对虚函数进行明确的实现,可以将它声明为纯虚函数。具体的实现由派生类完成。包含纯虚函数的类就是抽象类。**抽象类的一大特点是不能实例化**,也就是说,**抽象类只能作为基类被继承**,进而实例化派生类。

纯虚函数要在基类中进行声明,声明格式为:

virtual 返回值类型 纯虚函数名(参数表)=0;

纯虚函数的作用是在基类中为派生类保留一个函数的名字,它不具备函数的功能,不能被调用。**如果基类声明了纯虚函数,但是派生类没有对该纯虚函数进行定义,那么该函数变为派生类的纯虚函数,这意味着,这个派生类仍然为抽象类,不能实例化。**

因此,能进行实例化的类中必不能包含纯虚函数,即该类不能是抽象类。

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

class Base
{
public:
    virtual void show() = 0;
};

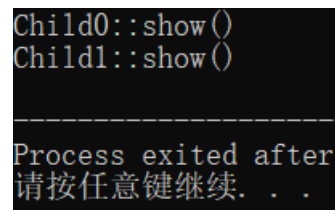
class Child0: public Base
{
public:
    void show(){cout << "Child0::show()" << endl;}
};

class Child1: public Base
{
public:
    void show(){cout << "Child1::show()" << endl;}
};

int main()
{
    Base *p;
    Child0 child0;
    Child1 child1;
    p = &child0;
    p->show();

    p = &child1;
    p->show();

    return 0;
}
```



```
Child0::show()
Child1::show()

-----
Process exited after
请按任意键继续. . .
```

在上面的示例中,Base 是一个抽象类,派生类 Child0 与 Child1 对纯虚函数 show() 进行了定义,因此,Child0 与 Child1 不再是抽象类,它们可以实例化。另外,示例中使用了基类对象指针,根据赋值兼容原则,它可以指向任何该基类的派生类,用基类对象指针调用不同派生类对象同名函数 show() 是虚函数的作用。不同对象对同一消息(函数调用)的不同响应正是多态性的体现。(消息的含义就是函数调用)

24. 输入/输出流