C++复习

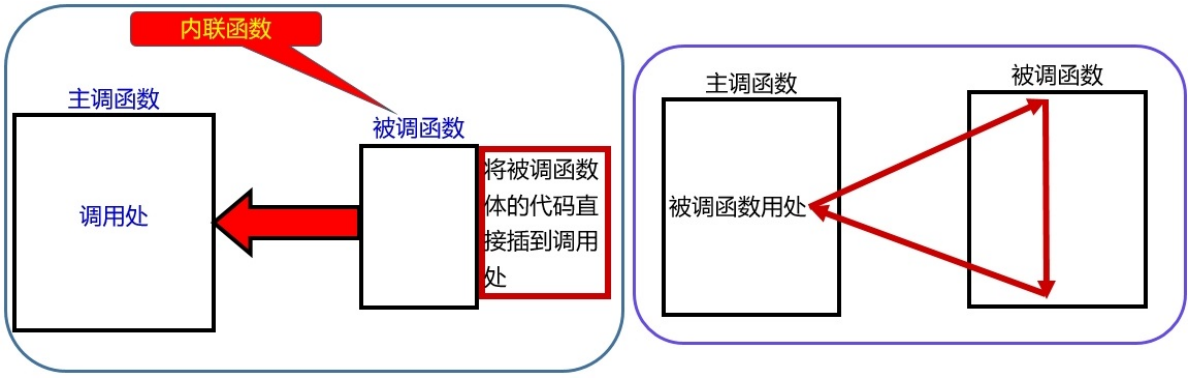
day1基础

头文件

```
#include <iostream>//c++包含头文件（输入输出流）
using namespace std;//引用命名空间
//cin  、 cout 输入输出

cin >> x;

cout << x << endl;

#include <iomanip> // 使用setw函数加的头文件\

cout <<"江苏省"<< swtw(5) <<"苏州市" <<endl;
```

内联函数

内联函数的实质是用存储空间（使用更多的存储空间）来换取时间（减少执行时间）。内联函数的定义方法是，在函数定义时，**在函数的类型前增加修饰词inline。**




```
inline void func(){
cout << "昆山市" << endl;
}
```

重载函数

所谓函数的**重载**是指完成不同功能的函数可以具有**相同的函数名。**C++的编译器是根据**函数的实参**来确定应该调用哪一个函数的。

1、定义的重载函数必须具有**不同的参数个数**，或**不同的参数类型。**只有这样编译系统才有可能根据不同的参数去调用不同的重载函数。

2、**仅返回值不同时，不能定义为重载函数。**即仅函数的类型不同，不能定义为重载函数。

```
int func(int x ,int y){
    return x + y;
}
```

```
int func(int x){
    return x;
}

double func(double x,double y){
    return x * y;
}

void main(){
    cout << "x = " << func(2) << endl;//调用func(int x)
    cont << "x + y =" << func(3,4) << endl;//调用func(int x ,int y)
    cont << "x * y =" << func(5.6,6.7) << endl;//调用func(double x,double y)
return 0;
}
```

day2指针基础

系统根据程序中定义变量的类型，给变量分配一定的长度空间。内存区的每个字节都有编号，称之为**地址**。

1. **基本数据类型**:

- `char`：1字节
- `short`：2字节
- `int`：4字节
- `long`：4字节（在某些平台上可能为8字节）
- `long long`：8字节
- `float`：4字节
- `double`：8字节
- `long double`：8字节或16字节（取决于平台）

2. **布尔类型**:

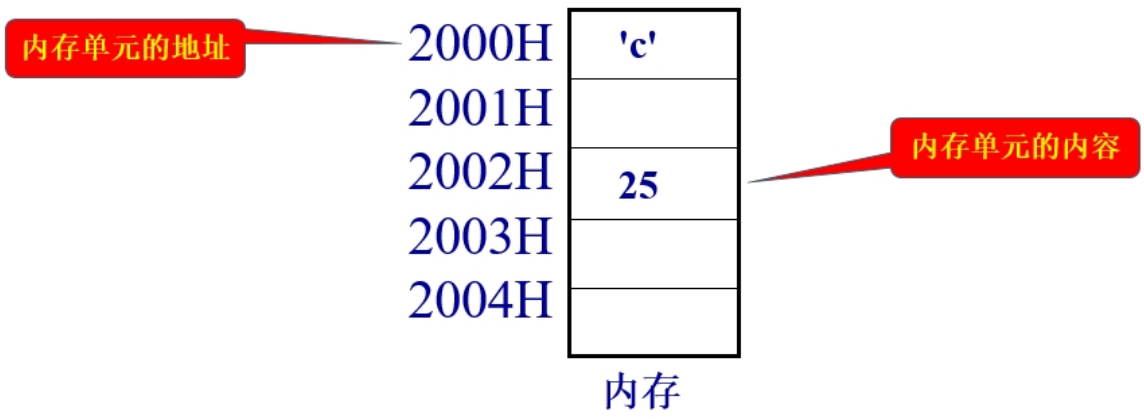
- `bool`：1字节（尽管它可能只能表示 true 或 false，但编译器通常为其分配1字节的内存）

3. **指针类型**:

- 指针大小通常与平台有关，32位系统上一般为4字节，64位系统上一般为8字节。

4. **用户自定义类型**（如结构体和类）：

- 用户自定义类型的内存占用大小是其成员变量内存大小之和，加上可能的对齐填充。



地址访问方式

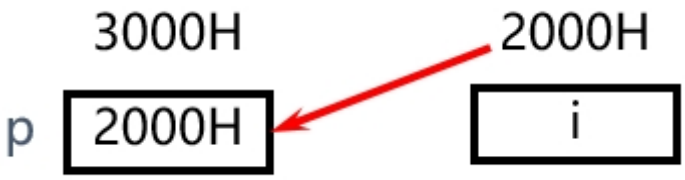
1. 直接访问

按变量地址存取变量的值。cin>>i；实际上放到定义 i 单元的**地址中**。

```
int i = 1;
cout << "i地址编号为"<< &i <<endl;
```

2. 间接访问

将变量的地址存放在另一个单元p中，通过 **p** 取出变量的地址，再针对变量操作。



一个变量的地址称为该变量的指针。

如果在程序中定义了一个变量或数组，那么，这个变量或数组的地址（指针）也就确定为一个**常量。**

```
//方法1先声明后定义
int i, *i_point;
i_point=&i;
//方法2声明时直接定义
int  i;
int *i_point=&i;
```

一个指针变量只能指向同一类型的变量。即整型指针变量只能放整型数据的地址，而不能放其它类型数据的地址。

* **在定义语句**中只表示变量的类型是指针，没有任何计算意义。

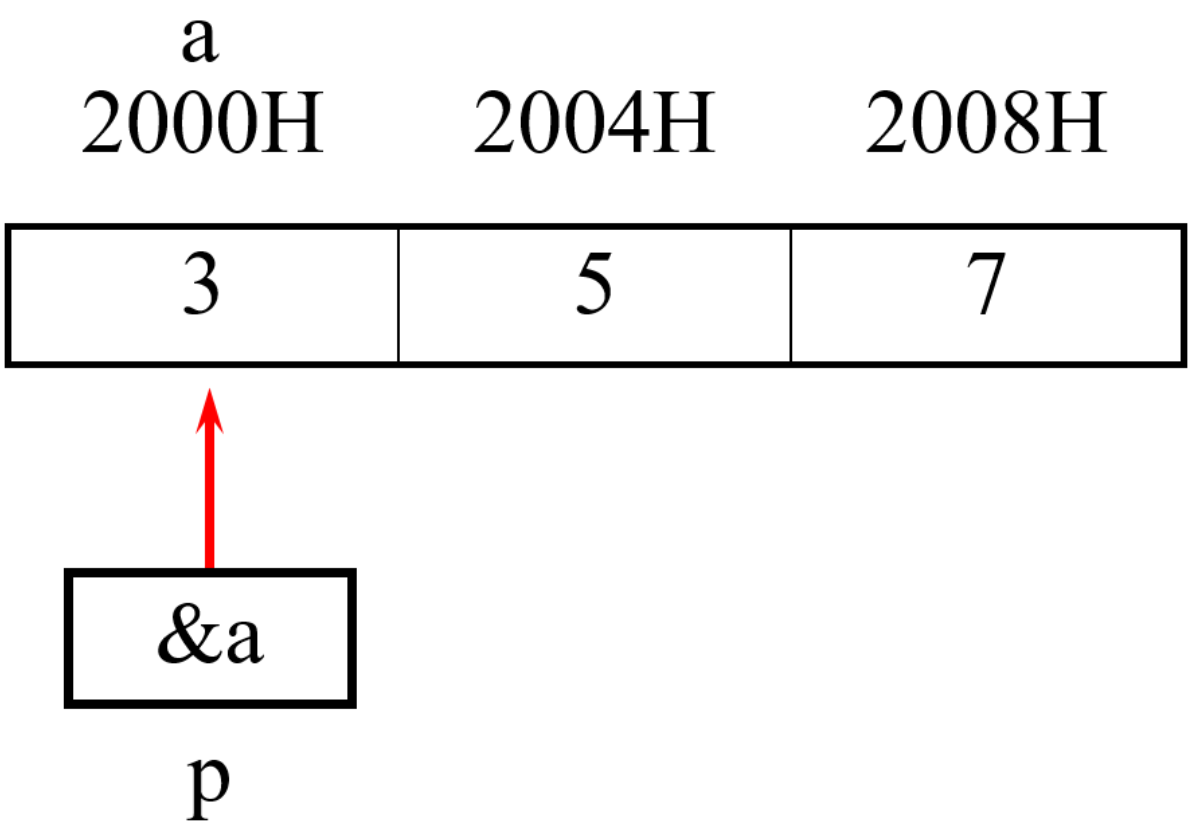
* **在语句中表示“指向”。&表示“地址”。**

```
int *p = Null;
int x = 2;
p = &x;
cout << "x地址编号为"<< &x <<endl;
cout << "p的值为"<< p <<endl;
//上面输出结果为一致的地址，下面输出结果为x的值
cout << "p地址里面的值为" << *p << endl;
```

```
//通过指针变量赋值
int  a, b;
int* p1, * p2;
p1 = &a;   p2 = &b;
*p1 = 10;  *p2 = 100;
cout << a << '\t' << b << endl;
cout << *p1 << '\t' << *p2 << endl;
return 0;
```

++, - -, * 优先级

优先级相同，都是右结合性

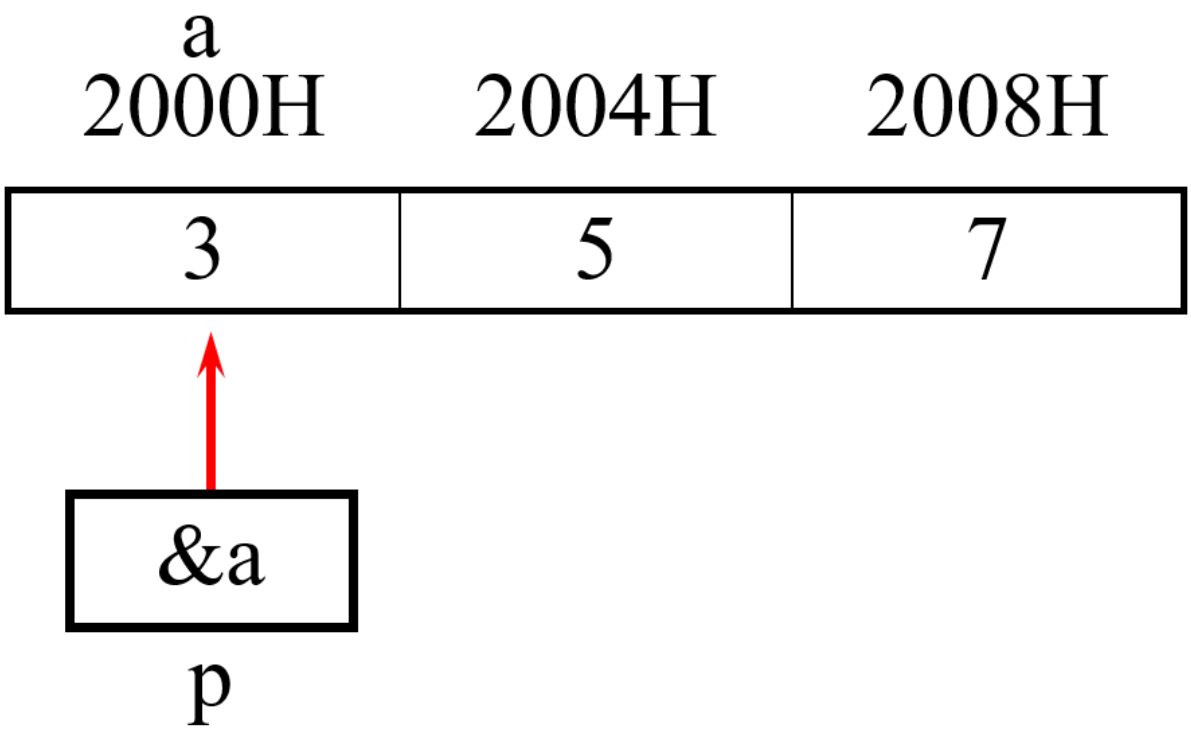


*int a=3, *p;*

p=&a;

*(*p)++;*

*相当于**a++**。表达式为3，a=4*



int a=3, *p;

p=&a;

***p++;**

*(p++)首先*p ， 然后p=p+1,指针指向下一个int单元 表达式为3, p=2004H。

++*p 等同与**++(*p)** 解引用后再++ 即 ***p=*p+1 a=4**

*++p 等同与 *(++p)，首先：p=p+1，然后取*p。即取p所指的下一个int单元的内容。表达式为5 p = 2004H 。

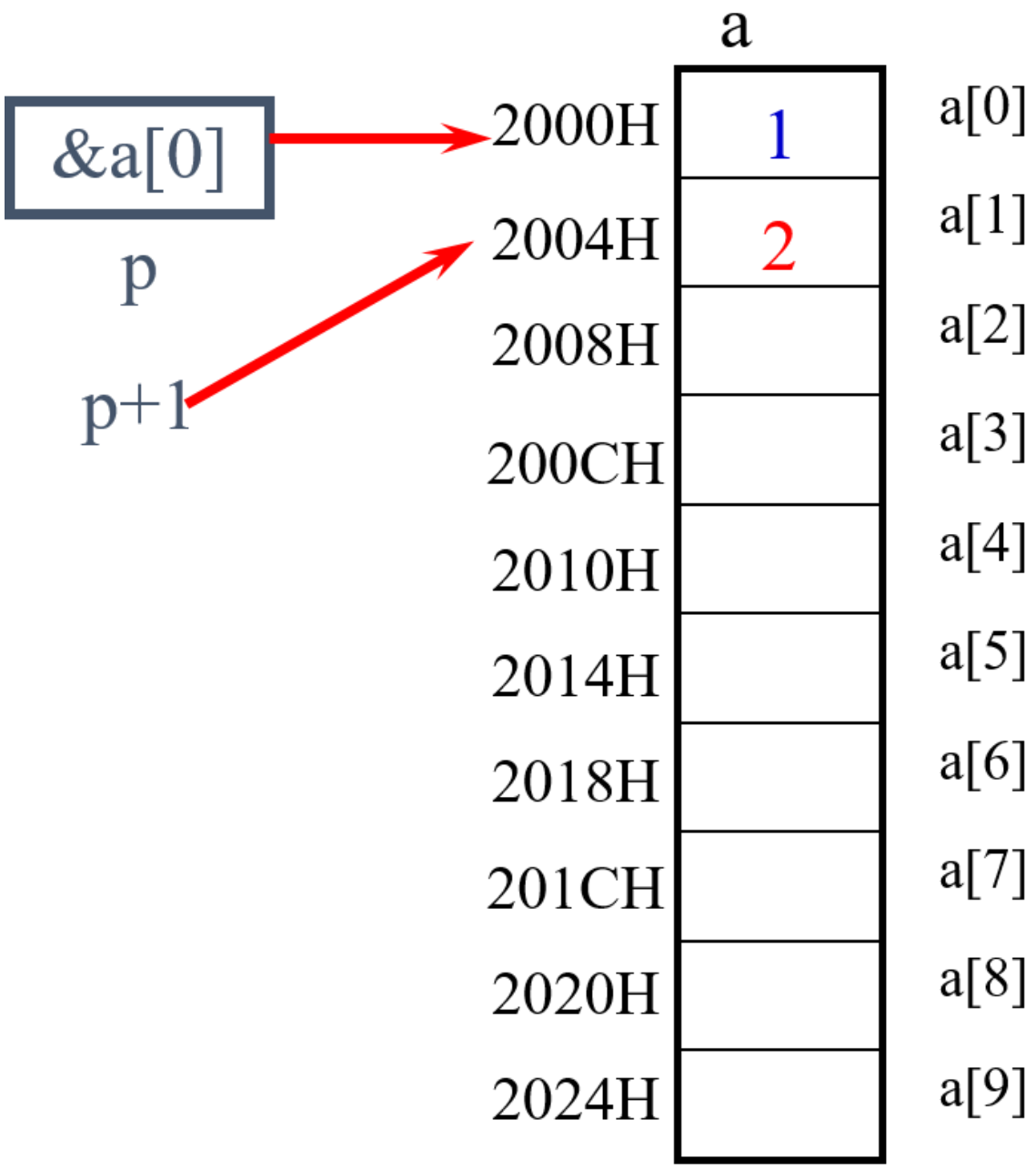
指针数据交换案例

```
void swapfunc(int* p1, int* p2)  // 将地址里面的值进行交换
{
    int temp=0;
    temp = *p1; // temp=5
    *p1 = *p2;
    *p2 = temp;
}
int main(){
    int a, b;
    int* pa, * pb;
    cout << "请输入a,b的值:";
    cin >> a >> b;

    pa = &a;
    pb = &b;
```

```
    if (a < b)
        swapfunc(pa, pb);
    cout << "a=" << a << ",b=" << b << endl;
    cout << *pa << "," << *pb << endl;
    return 0;
}
```

指针数组

```
int a[5] = { 34,56,32,89,19 };
int* pa = a; // 将数组的首地址赋给指针变量pa  pa=&a[0]
cout << "\n通过循环输出数组a元素值:\n";
for (int i = 0; i < 5; i++)
    cout << *(pa + i) << "  ";
cout << endl << endl;
cout << "a[0]=" << *pa << endl;
*pa = 59;
cout << "a[0]=" << *pa << endl;
cout << "结果为:"<<*(pa + 1) << endl;//pa+1是指向数组的下一个元素，而不是下一个字节
```



指针引用

对变量取另外一个名字 (外号)，这个名字称为该变量的引用。

<类型> &<引用变量名> = <原变量名>;

其中原变量名必须是一个已定义过的变量。如:

```
int  max ;
int  &refmax=max;
```

refmax并没有重新在内存中开辟单元，只是**引用**max的单元。max与refmax**在内存中占用同一地址**，即同一地址两个名字。

对引用类型的变量，说明以下几点:

1、引用在定义的时候要初始化。

2、对引用的操作就是对被引用的变量的操作。

3、 引用类型变量的初始化值不能是一个常数。

4、引用和变量一样有地址，可以对其地址进行操作，即将其地址赋给一指针。

```
int  a, *p;

int &m=a;

p=&m;

*p=10;
```

5、可以用动态分配的内存空间来初始化一个引用变量。

```
float  &reff = * new float ;     // 用new开辟一个空间，取一个别名reff
    reff= 200;           // 给空间赋值
    cout << reff ;           // 输出200
    delete &reff;            // 收回这个空间
```

这个空间只有别名，但程序可以引用到。

```
float  *p,  a;
p=new  float;
float  a=* new  float;//这个是错误的
```

6、指针与引用的区别：

①指针是通过地址**间接**访问某个变量，而引用是通过别名**直接**访问某个变量。

②引用必须初始化，而**一旦被初始化后不得再作为其它变量的别名。**

当&a的前面有**类型符**时（如int &a），它必然是对引用的声明；如果前面**无类型符**（如cout<<&a）,则是取变量的地址。

以下的声明是非法的

①企图建立数组的引用　int & a[9];

②企图建立指向引用的指针　int & *p;

③企图建立引用的引用　int & &px;

7、对常量（用const声明）的引用

```
void main(void)
{
    const int &r=8; //说明r为常量，不可赋值
    cout<<"r="<<r<<endl;
    //  r+=15;       //r为常量，不可作赋值运算
    cout<<"r="<<r<<endl;
}
```

8、引用与函数

引用的用途主要是用来作函数的参数或函数的返回值。

引用作函数的形参，实际上是在被调函数中对实参变量进行操作。

```
void  change(int &x,  int &y)   // x,y是实参a,b的别名
{   int   t;
    t=x;
    x=y;
    y=t;
}
void main(void)
{   int  a=3,b=5;
    change(a,b);  //实参为变量
    cout<<a<<'\t'<<b<<endl;
}
```

9、函数的返回值为引用类型

可以把函数定义为引用类型，这时函数的返回值即为某一变量的引用（别名），**因此，它相当于返回了一个变量，所以可对其返回值进行赋值操作。这一点类同于函数的返回值为指针类型 。**

```
#include <iostream>
using namespace std;
int  a = 4;
int& f(int  x)//函数返回a的引用，即a的别名
{
    a = a + x;
    return  a;
}
int main(int argc, char* argv[])
{
    int   t = 5;
    cout << f(t) << endl;
    f(t) = 20;
    cout << f(t) << endl;
    t = f(t);
    cout << f(t) << endl;
    return 0;
}
```

一个函数返回引用类型，**必须返回某个类型的变量。**

```
    语句：getdata()=8;

    就相当于int  &temp=8;

                    temp=8 ;
```

注意：

由于函数调用返回的引用类型是在函数运行结束后产生的，所以**函数不能返回自动变量和形参。**返回的变量的引用，**这个变量必须是全局变量或静态局部变量，即存储在静态区中的变量。**

我们都知道，函数作为一种程序实体，它有名字、类型、地址和存储空间，一般说来**函数不能作为左值**（即函数不能放在赋值号左边）。但如果将函数定义为返回引用类型，因为返回的是一个变量的别名，就可以将函数放在左边，即给这个变量赋值。

const型指针

const类型变量

当用const限制说明标识符时，表示所说明的数据类型为常量类型。可分为const型常量和const型指针。

可用const限制定义标识符量，如：

```
const  int MaxLine =1000;
const  float  Pi=3.1415926
```

用const定义的标识符常量时，一定要对其初始化。**在声明时进行初始化是对这种常量置值的唯一方法**，不能用赋值运算符对这种常量进行赋值。如：

```
MaxLine = 35; //错误
```

1)禁写指针

声明语句格式为： 数据类型 * const 指针变量名

如：int r=6;

int * const pr=&r;

则指针pr被禁写，即**pr将始终指向一个地址，成为一个指针常量。**它将不能再作为左值而放在赋值号的左边。（举例说明）

同样，禁写指针一定要在定义的时候赋初值。虽然指针被禁写，但其间接引用并没有被禁写。即可以通过pr对r赋值。*pr=8;

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

int main(int argc, char* argv[])
{
    int  a, b;
    int* const  pa = &a;     // 一定要赋初值，pa是常量，不能在程序中//被改变
    *pa = 10;        // 可以间接引用
    pa = &b;             // 非法，pa为常量

    return 0;
}
```

2）禁写间接引用

声明语句格式如下：

const 数据类型 *指针变量名;

所声明的指针指向一禁写的实体，即间接引用不能被改写。如： const int *p;。 所以程序中不能出现诸如 *p= 的语句，但指针p并未被禁写，因而可对指针p进行改写。

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

```
int main(int argc, char* argv[])
{
    int  a = 3, b = 5;
    const  int* pa = &b;     //可以不赋初值
    pa = &a;          //指针变量可以重新赋值
    cout << *pa << endl;    //输出3
  //  *pa = 10;       //非法，指针指向的内容不能赋值
    a = 100;          //变量可以重新赋值
    cout << *pa << endl;  //输出100

    return 0;
}
//即不可以通过指针对变量重新赋值
```

3)禁写指针又禁写间接引用

将上面两种情况结合起来，声明语句为下面的格式

const 数据类型 *const 指针变量名

如：const int *const px=&x

说明：px是一个指针常量，它指向一禁写的实体，并且指针本身也被禁写。

诸如：px= *px= 此类的语句都是非法的。

在定义时必须赋初值。

```
    //int x = 1298;
    //int* const px = &x;
    //cout << "第一次输出结果:" << endl;
    //cout << x << endl;
    //cout << *px << endl;
    //*px = 1234;//非法 x值不可以改变，但是px指针变量指向的地址是可以的

    //cout << "第二次输出结果:" << endl;
    //x = 500;
    //cout << x << endl;
    //cout << *px << endl;
    //int y = 888;
    // px = &y; // 非法 px指针变量指向的地址是不能改变，但是原来的x的值是可以的。

    int x = 1298;
    const int* const px = &x;//px指针和x值都是不可以改变的
    cout << "第一次输出结果:" << endl;
    cout << x << endl;
    cout << *px << endl;
```

day3类与对象

面向对象

面向对象编程（Object Oriented Programming，OOP，面向对象程序设计）的主要思想是把构成问题的各个事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙一个事物在整个解决问题的步骤中的行为。面向对象程序设计中的概念主要包括：对象、类、数据抽象、继承、动态绑定、数据封装、多态性、消息传递。通过这些概念面向对象的思想得到了具体的体现。

（一）类的实现

在开发过程中，类的实现是核心问题。在用面向对象风格所写的系统中，所有的数据都被封装在类的实例中。而整个程序则被封装在一个更高级的类中。在使用既存部件的面向对象系统中，可以只花费少量时间和工作量来实现软件。只要增加类的实例，开发少量的新类和实现各个对象之间互相通信的操作，就能建立需要的软件。

（二）应用系统的实现

应用系统的实现是在所有的类都被实现之后的事。实现一个系统是一个比用过程性方法更简单、更简短的过程。有些实例将在其他类的初始化过程中使用。而其余的则必须用某种主过程显式地加以说明，或者当作系统最高层的类的表示的一部分。

在C++和C中有一个main（）函数，可以使用这个过程来说明构成系统主要对象的那些类的实例。

（三）面向对象测试

⑴算法层。

⑵类层。（测试封装在同一个类中的所有方法和属性之间的相互作用。）

⑶模板层。（测试一组协同工作的类之间的相互作用。）

⑷系统层。（把各个子系统组装成完整的面向对象软件系统，在组装过程中同时进行测试。）

类与对象（一）

1、【案例分析】

a、按钮对象：

按钮的内容、大小，按钮的字体、图案等等

针对按钮的各种操作，创建、单击、双击、拖动等

b、班级对象：

班级的静态特征，所属的系和专业、班级的人数，所在的教室等。这种静态特征称为属性；

班级的动态特征，如学习、开会、体育比赛等，这种动态特征称为行为。

任何一个对象都应当具有这两个要素，一是属性(attribute)；二是行为(behavior)，即能根据外界给的信息进行相应的操作。对象是由一组属性和一组行为构成的。

面向对象的程序设计采用了以上人们所熟悉的这种思路。使用面向对象的程序设计方法设计一个复杂的软件系统时，**首要的问题是确定该系统是由哪些对象组成的，并且设计这些对象。**在C++中，每个对象都是由数据和函数(即操作代码)这两部分组成的。

程序设计者的任务包括两个方面：一是设计所需的各种类和对象，即决定把哪些数据和操作封装在一起；二是考虑怎样向有关对象发送消息，以完成所需的任务。**各个对象的操作完成了，整体任务也就完成了。**

因此人们设想把相关的数据和操作放在一起，形成一个整体，与外界相对分隔。这就是面向对象的程序设计中的对象。

在**面向过程**的结构化程序设计中，人们常使用这样的公式来表述程序：

程序=算法 + 数据结构

面向对象的程序组成：

对象 = 算法 + 数据结构

程序=(对象+对象+对象+.....)+ 消息

消息的作用就是对对象的控制。

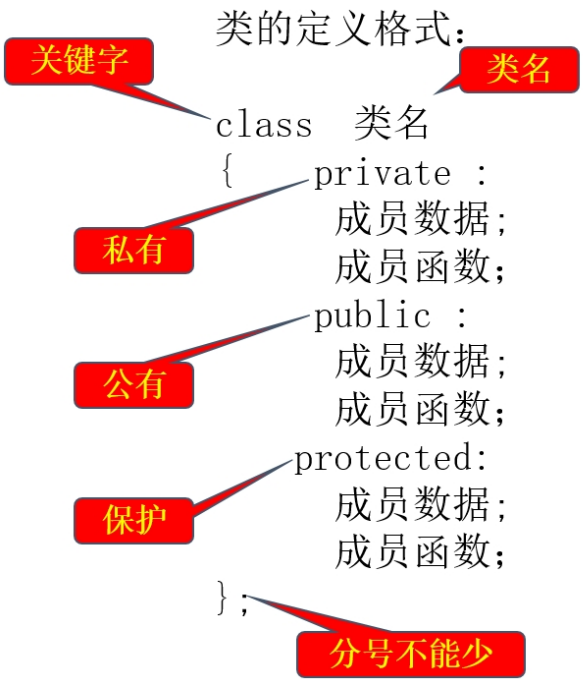
程序设计的关键是设计好每一个对象以及确定向这些对象发出的命令，使各对象完成相应的操作。

每一个实体都是对象。有一些对象是具有相同的结构和特性的。 每个对象都属于一个特定的类型。在C++中对象的类型称为类(class)。**类代表了某一批对象的共性和特征。 类是对象的抽象，而对象是类的具体实例(instance)。**

2、【类的定义】

类是一种复杂的数据类型，它是将**不同类型的数据**和**与这些数据相关的运算**封装在一起的集合体。

类将一些数据及与数据相关的**函数**封装在一起，使类中的数据得到很好的“保护”。在大型程序中不会被随意修改。




```
class Student
{ private :
    char Name[20];
    float Math;
    float Chiese;
  public :
    float average;
    void SetName(char *name);
    void SetMath(float math);
    void SetChinese(float ch);
    float GetAverage(void);
};
```

用关键字**priviate**限定的成员称为**私有成员**，对私有成员限定在该类的内部使用，**即只允许该类中的成员函数使用私有的成员数据，**对于私有的成员函数，只能被**该类内的成员函数调用**；类就相当于私有成员的作用域。

用关键字**public**限定的成员称为**公有成员**，公有成员的数据或函数不受类的限制，**可以在类内或类外自由使用；**对类而言是透明的。

而用关键字**protected**所限定的成员称为**保护成员，**只允许在类内及该类的派生类中使用保护的数据或函数。即保护成员的作用域是**该类及该类的派生类。**

```
// 设计一个学生类
class CStudent
{
public:  // 公有成员
    void InputData()// 类内部声明函数
    {
        cout << "\n\n请输入学号:";
        cin >> sno;
        cout << "请输入姓名:";
        cin >> sname;
        cout << "请输入分数:";
```

```
        cin >> dscore;
    }
    void OutputData(); // 声明函数
    double daverage; // 平均分

private: // 私有成员
    int sno; // 学号
    char sname[10]; // 姓名
    double dscore; // 分数

protected:  // 保护成员

};
```

	私有函数	公有函数	保护函数
类内函数	可以调用	可以调用	可以调用
类内函数	不可调用	可以调用	不可调用

每一个限制词(private等)在类体中可使用多次。一旦使用了限制词，该限制词一直有效，直到下一个限制词开始为止。

如果未加说明，类中成员默认的访问权限是private，即私有的。

在类体外定义成员函数的格式：

```
<type>   < class_name > :: < func_name > (<参数表>)
{
......  //函数体
}
```

```
// 在类的外面函数实现：加（类名::）
//在类体外定义时使用内联函数，成员函数前加上关键字inline，可以节省程序调用函数的时间
inline void CStudent::OutputData()
{
    cout << "\n\n学生数据信息如下:" << endl;
    cout << "学号:" << sno << endl;
    cout << "姓名:" << sname << endl;
    cout << "分数:" << dscore << endl;
}
```

3、【对象及使用】

一个对象的成员就是该对象的类所定义的成员，有成员数据和成员函数，**引用时同结构体变量类似，用“.”运算符。**

用成员选择运算符“.”只能访问对象的**公有成员**，而不能访问对象的私有成员或保护成员。**若要访问对象的私有的数据成员，只能通过对象的公有成员函数来获取。**

4、【类作用域、类类型的作用域和对象的作用域 】

类体的区域称为**类作用域。**类的成员函数与成员数据，其作用域都是属于类的作用域，仅在该类的范围内有效，**故不能在主函数中直接通过函数名和成员名来调用函数。**

类类型的作用域：在函数定义之外定义的类，其类名的作用域为文件作用域；而在函数体内定义的类，其类名的作用域为块作用域 。

> **对象的作用域**与前面介绍的变量作用域完全相同 ， **全局对象、局部对象、局部静态对象**等。

```
//整体代码如下，可以直接拷贝验证
#include <iostream>
using namespace std;

// 设计一个学生类
class CStudent
{
public:  // 公有成员
    void InputData()
    {
        cout << "\n\n请输入学号:";
        cin >> sno;
        cout << "请输入姓名:";
        cin >> sname;
        cout << "请输入分数:";
        cin >> dscore;
    }
    void OutputData();  // 声明函数
    double daverage; // 平均分

private: // 私有成员
    int sno; // 学号
    char sname[10]; // 姓名
    double dscore; // 分数

protected:  // 保护成员

};

CStudent stu2;  // 声明全局对象

// 函数实现在外面加：（类名::）
inline void CStudent::OutputData()
{
    cout << "\n\n学生数据信息如下:" << endl;
    cout << "学号:" << sno << endl;
    cout << "姓名:" << sname << endl;
    cout << "分数:" << dscore << endl;
}

void testfunc()
{
    stu2.daverage = 100;
    cout << "小明的平均分为:" << stu2.daverage << endl << endl;
}

int main()
```

```
{
    CStudent stu1;
    stu1.InputData(); // 调用输入函数为私有数据成员赋值
    stu1.OutputData(); // 调用输出函数将数据信息输出
    stu1.daverage = 100; // 可以方位davergae，因为这个数据类型是公有的
    // stu1.dscore = 90; // 不可以访问因为dscore它是私有数据成员，外界是不可以访问

    testfunc();

    return 0;
}
```

类与对象（二）

1、类的嵌套

在定义一个类时, 在其类体中又包含了一个类的完整定义，称为类的嵌套 。**类是允许嵌套定义的 。**

```
class CC1
{
public:
    int x;
    void Func();

    class CC2
    {
    public:
        int x;
        void Func();
    }objc;
};
void CC1::Func()
{
    x = 3000;
    cout << "x=" << x << endl;
}
//类嵌套外部函数定义
void CC1::CC2::Func()
{
    x = 40000;
    cout << "x=" << x << endl;
}

int main()
{
    class CC1 obj;
    obj.Func();
    obj.objc.Func();//访问类嵌套的成员函数

    cout << "\n\n";
    cout << "2:x=" << obj.x << endl;//输出3000
    cout << "2:x=" << obj.objc.x << endl;//输出40000

    return 0;
}
```

2、对象引用私有数据成员

1）通过公有函数为私有成员赋值

```
// 1公有函数为私有成员赋值
class CA
{
private:
    float x, y;
public:
    void setab(float a, float b) {
        x = a;
        y = b;
    }
    void readab() {
        cout << "x=" << x << endl;
        cout << "y=" << y << endl;
    }
};
int main()
{
    class CA obj;
    obj.setab(2.3, 3.4);
    obj.readab();

    return 0;
}
```

2）利用指针访问私有数据成员

```
// 2利用指针访问私有数据成员
class CTest {
    int x, y;
public:
    void setxy(int a, int b) {
        x = a;
        y = b;
    }
    void printxy() {
        cout << "\nx=" << x << ",y=" << y << endl << endl;
    }

    void getxy(int* px, int* py) // 提取x y的值
    {
        *px = x;
        *py = y;
    }
};

int main()
{
    CTest obj;
    obj.setxy(100, 200);
    obj.printxy();
```

```
    int m, n;
    obj.getxy(&m, &n); // 将m=x,n=y
    cout << "\n\nm=" << m << ",n=" << n << endl << endl;

    return 0;
}
```

3）利用函数访问私有数据成员

```
// 3利用函数访问私有数据成员
class CTest {
    int x, y;
public:
    void setxy(int a, int b) {
        x = a;
        y = b;
    }
    void printxy() {
        cout << "\nx=" << x << ",y=" << y << endl << endl;
    }

    int getx() { return x; }
    int gety() { return y; }
};

int main()
{
    CTest obj;
    obj.setxy(100, 200);
    obj.printxy();

    cout << "\n\nx=" << obj.getx() << ",y=" << obj.gety() << endl << endl;

    return 0;
}
```

4）利用引用访问私有数据成员

```
// 4利用引用访问私有数据成员
class CTest {
    int x, y;
public:
    void setxy(int a, int b) {
        x = a;
        y = b;
    }
    void printxy() {
        cout << "\nx=" << x << ",y=" << y << endl << endl;
    }
    void getxy(int& px, int& py) {
        px = x;
        py = y;
    }
};
```

```
int main()
{
    CTest obj;
    obj.setxy(100, 200);
    obj.printxy();

    int m, n;
    obj.getxy(m, n);
    cout << "m=" << m << ",n=" << n << endl << endl;

    return 0;
}
```

3、成员函数的重载

类中的成员函数与前面介绍的普通函数一样，成员函数可以**带有缺省的参数**，也可以**重载**成员函数。重载时，函数的形参必须在类型或数目上不同。

缺省参数的成员函数

```
class CTest
{
    int x, y;
    int m, n;
public:
    void setxy(int a, int b)
    {
        x = a;
        y = b;
    }
    void setxy(int a, int b, int c, int d)
    {
        x = a;
        y = b;
        m = c;
        n = d;
    }

    void dispxy(int x)
    {
        cout << x << "," << y << endl << endl;
    }
    void dispxymn()
    {
        cout << x << "," << y << "," << m << "," << n << endl << endl;
    }
};
int main()
{
    CTest obj1, obj2;

    // 参数不同
    obj1.setxy(10, 20);
    obj2.setxy(10, 20, 30, 40);
```

```
    // 参数类型不同
    obj1.dispxy(666);
    obj2.dispxymn();

    return 0;
}
```

定义类的指针及如何用指针来引用对象

定义类的数组及数组中元素的引用

```
// 指针引用对象
class CTest
{
public:
    double sum()
    {
        return x + y;
    }
    void setxy(double a, double b) {
        x = a;
        y = b;
    }
    void dispxy()
    {
        cout << "x=" << x << "," << "y=" << y << endl << endl;
    }

private:
    int x, y;

};
int main()
{
    CTest obj1, obj2;  // 定义对象
    CTest* pobj; // 定义类的指针（对象指针）
    pobj = &obj1;

    pobj->setxy(3.4, 5.6); // 通过指针引用对象的成员函数
    pobj->dispxy();

    cout << "x+y=" << pobj->sum() << endl;

    return 0;
}
```

4、this指针

不同对象占据内存中的不同区域，它们所保存的数据各不相同，但**对成员数据进行操作的成员函数的程序代码均是一样的。**

当对一个对象调用成员函数时，**编译程序先将对象的地址赋给this指针，然后调用成员函数**，每次成员函数存取数据成员时，也隐含使用this指针。

```
class CTest
{
private:
    int x;
public:
    int getx() const {
        return x;
    }
    void setx(int x) {
        this->x = x;
        cout << "this指针存储的内存地址为:" << this << endl << endl;
    }
};

int main()
{
    CTest obj;
    obj.setx(888);
    cout << "对象obj在内存的地址为:" << &obj << endl;
    cout << "对象obj所保存的值为:" << obj.getx() << endl;
    cout << endl;

    return 0;
}
```

类的其他特性

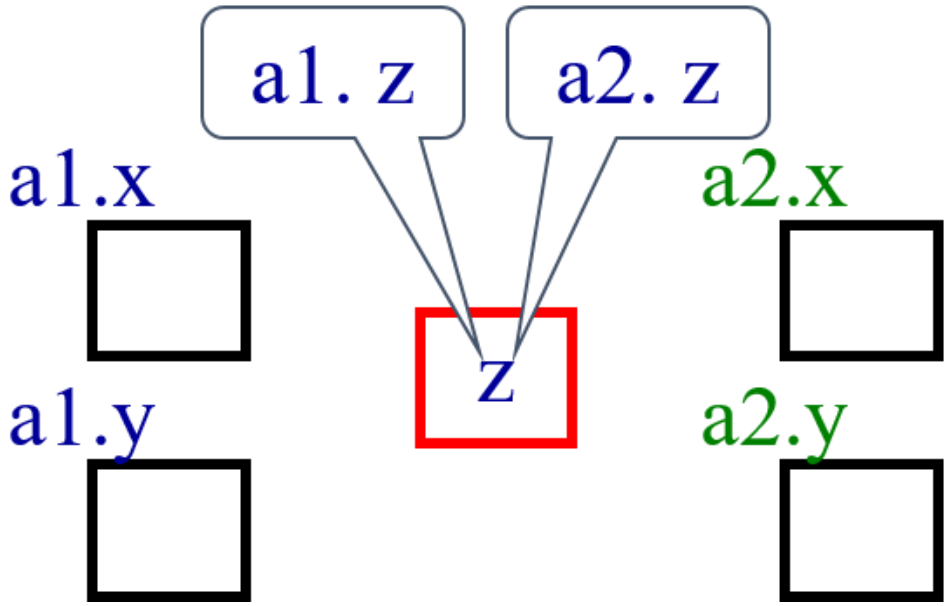
一、静态成员函数

通常，每当说明一个对象时，把该类中的有关成员数据拷贝到该对象中，即同一类的不同对象，**其成员数据之间是互相独立的。**

当我们将类的某一个**数据成员的存储类型指定为静态类型时**，则由该类所产生的所有对象，其静态成员均共享**一个存储空间**，**这个空间是在编译的时候分配的。**换而言之，在说明对象时，并不为静态类型的成员分配空间。在类定义中，用关键字static修饰的数据成员称为静态数据成员。

```
class A{
    int x,y;  static  int z;
    public:
    void Setxy(int a, int b)
    {  x=a;   y=b;}
};
```

不同对象，同一空间



有关静态数据成员的使用，说明以下几点：

1、类的静态数据成员是**静态分配存储空间**的，而其它成员是动态分配存储空间的（全局变量除外）。当类中没有定义静态数据成员时，在程序执行期间遇到说明类的对象时，才为对象的所有成员依次分配存储空间，这种存储空间的分配是动态的；而当类中定义了静态数据成员时，**在编译时，就要为类的静态数据成员分配存储空间。**

2、**必须在文件作用域中，对静态数据成员作一次且只能作一次定义性说明。**因为静态数据成员在定义性说明时已分配了存储空间，所以通过静态数据成员名前加上类名和作用域运算符，可直接引用静态数据成员。在C++中，静态变量缺省的初值为0，所以静态数据成员总有唯一的初值。当然，**在对静态数据成员作定义性的说明时，也可以指定一个初值。**

```
class CTestA
{
public:
    CTestA(int a, int b)//构造函数--函数名和类目一样
    {
        i = a;
        j = b;
    }

    void dispijxy()
    {
        cout << "\ni=" << i << ",j=" << j << ",x=" << x << ",y=" << y << endl <<
endl;
    }

private:
    int i, j;
    static int x, y;  // 定义静态数据成员
};
```

```
// 静态成员定义性说明
int CTestA::x = 5;
int CTestA::y = 5;

int main()
{
    CTestA obj1(10, 20);
    obj1.dispijxy();//输出i=10,j=20,x=5,y=5

    CTestA obj2(100, 200);
    obj2.dispijxy();//输出i=100,j=200,x=5,y=5

    return 0;
}
```

3、静态数据成员具有全局变量和局部变量的一些特性。静态数据成员与全局变量一样都是静态分配存储空间的，**但全局变量在程序中的任何位置都可以访问它，而静态数据成员受到访问权限的约束。必须是public权限时，才可能在类外进行访问。**

4、**为了保持静态数据成员取值的一致性，通常在构造函数中不给静态数据成员置初值，而是在对静态数据成员的定义性说明时指定初值。**

```
class CTest
{
public:
    CTest(int x = 0)//构造函数，初始化x
    {
        i = x;
        icount++;
    }
    void disp()
    {
        cout << "i=" << i << ",icount=" << icount << endl;
    }

private:
    int i;
    static int icount;//通常在构造函数中不给静态数据成员置初值
};

int CTest::icount = 0;//在对静态数据成员的定义性说明时指定初值

int main()
{
    CTest obj1[100];//声明一个对象数组，拥有100个对象
    CTest obj2[10];//声明一个对象数组，拥有10个对象
    obj1->disp();//输出i = 0,icount = 110

    return 0;
}
```

【静态成员函数】

可以将类的成员函数定义为静态的成员函数。即使用关键字static来修饰成员函数 。

```
class A
{    float x, y;
public :
     A( ){  }
     static   void sum(void)  { ..... }
};
```

对静态成员函数的用法说明以下几点：

1、**与静态数据成员一样，在类外的程序代码中，通过类名加上作用域操作符，可直接调用静态成员函数。**

2、**静态成员函数只能直接使用本类的静态数据成员或静态成员函数，但不能直接使用非静态的数据成员**（可以引用使用）。这是因为静态成员函数可被其它程序代码直接调用，所以，**它不包含对象地址的this指针。**

```
class CTest
{
public:
    CTest(int a)
    {
        x = a;
        y = y + x;
    }
    static void disp(CTest obj)
    {
        cout << "x=" << obj.x << ",y=" << y << endl;
    }

private:
    int x;
    static int y; // 静态数据成员
};

int CTest::y=10;

int main()
{
    CTest obj1(5), obj2(10);

    CTest::disp(obj1);
    CTest::disp(obj2);

    return 0;
}
```

3、**静态成员函数的实现部分在类定义之外定义时，其前面不能加修饰词static。**这是由于关键字static不是数据类型的组成部分,因此，在类外定义静态成员函数的实现部分时,不能使用这个关键字

4、**不能把静态成员函数定义为虚函数。**静态成员函数也是在编译时分配存储空间，所以在程序的执行过程中不能提供多态性。

5、**可将静态成员函数定义为内联的（inline）**，其定义方法与非静态成员函数完全相同。

二、指向类的指针

一个指向 C++ 类的指针与指向结构的指针类似，访问指向类的指针的成员，需要使用成员访问运算符 **->**，就像访问指向结构的指针一样。与所有的指针一样，您必须在使用指针之前，对指针进行初始化。

```
int main()
{
    CTest obj1(5), obj2(10);

    CTest::disp(obj1);
    CTest::disp(obj2);

    CTest* pobj;
    pobj = &obj1;
    pobj->disp(obj1);

    return 0;
}
```

day4类与对象二

构造函数

构造函数和析构函数是在类体中说明的**两种特殊的成员函数。**构造函数是在创建对象时，使用给定的值来将对象初始化。

析构函数的功能正好相反，是在系统释放对象前，对对象做一些善后工作。构造函数**可以带参数、可以重载**，同时没有返回值。

构造函数是类的成员函数，系统约定构造函数名必须与类名相同。**构造函数提供了初始化对象的一种简单的方法。**

【构造函数说明】

1.**构造函数的函数名必须与类名相同。**构造函数的主要作用是完成初始化对象的数据成员以及其它的初始化工作。

2.在定义构造函数时，**不能指定函数返回值的类型，也不能指定为void类型。**

3.一个类可以定义若干个构造函数。当定义多个构造函数时，**必须满足函数重载的原则。**

4.构造函数可以指定参数的缺省值。

5.若定义的类要说明该类的对象时，构造函数必须是**公有的成员函数。**如果定义的类仅用于派生其它类时，则可将构造函数定义为**保护的成员函数。**

由于构造函数属于类的成员函数，它对私有数据成员、保护的数据成员和公有的数据成员均能进行初始化。

对局部对象，静态对象，全局对象的初始化对于局部对象，每次定义对象时，都要调用构造函数。**对于静态对象，是在首次定义对象时，调用构造函数的，且由于对象一直存在，只调用一次构造函数。**

> 对于全局对象，是**在main函数执行之前调用构造函数的。**

```cpp
#include <iostream>
using namespace std;
class CTestA
{
public:
    CTestA()
    {
        x = 0;
        y = 0;
        cout << "\n初始化静态局部对象." << endl;
    }
    CTestA(double a)
    {
        x = a;
        y = 0;
        cout << "初始化全局对象." << endl;
    }
    CTestA(double a,double b)
    {
        x = a;
        y = b;
        cout << "初始化局部对象." << endl;
    }
private:
    double x , y ;
};

CTestA objb(88.8);  // 全局对象初始优先级最高，排行在第一

void FuncTest()
{
    cout << "\n程序开始进入FuncTest()函数.\n\n";
    CTestA objc(10, 20);
    static CTestA objd; // 初始化局部静态对象
}
int main()
{
    cout << "\n程序开始执行-->main()函数\n\n";
    CTestA obja(4.5, 6.5); // 定义局部对象
    FuncTest();
    return 0;
}
//输出结果
/*
初始化全局对象.

程序开始执行-->main()函数

初始化局部对象.

程序开始进入FuncTest()函数.

初始化局部对象.

初始化静态局部对象.
```

```
*/
```

【缺省构造函数】

在定义类时，若没有定义类的构造函数，则编译器自动产生一个缺省的构造函数，其格式为：

```
className::className() { }
```

缺省的构造函数并不对所产生对象的数据成员赋初值；即**新产生对象的数据成员的值是不确定的。**

【关于缺省的构造函数】，说明以下几点：

1、在定义类时，只要**显式**定义了一个类的构造函数，则编译器就不产生缺省的构造函数。

2、所有的对象在定义时，必须调用构造函数，不存在没有构造函数的对象！

3、在类中，若定义了没有参数的构造函数，或各参数均有缺省值的构造函数也称为缺省的构造函数，**缺省的构造函数只能有一个。**

4、产生对象时，系统必定要调用构造函数。**所以任一对象的构造函数必须唯一。**

```
#include <iostream>
using namespace std;
class CTestA
{
public:
    CTestA()
    {
        cout << "调用缺省构造函数.\n";
    }
    void setxy(int a, int b)
    {
        x = a;
        y = b;
    }
    void disp()
    {
        cout << "x=" << x << "," << "y=" << y << "\n";
    }
private:
    double x, y;
};
int main()
{
    CTestA obja, objb; // 产生对象时候，自动调用缺省的构造函数，不赋值
    obja.setxy(40, 90);
    cout << "\nobja对象结果为:\n";
    obja.disp();

    cout << "\nobjb对象结果为:\n";
    objb.disp();
    return 0;
}
//输出结果
```

```
/*
调用缺省构造函数.
调用缺省构造函数.

obja对象结果为:
x=40,y=90

objb对象结果为:
x=-9.25596e+61,y=-9.25596e+61
*/
```

【构造函数与new运算符】

可以使用new运算符来动态地建立对象。建立时**要自动调用构造函数**，以便完成初始化对象的数据成员。最后返回这个动态对象的起始地址。

用new运算符产生的动态对象，在不再使用这种对象时，必须用delete运算符来释放对象所占用的存储空间。

用new建立类的对象时，可以使用参数初始化动态空间。

```
#include <iostream>
using namespace std;
class CTestA
{
public:
    CTestA()  // 缺省构造函数
    {   x = 0;
        y = 0;
        cout << "调用缺省构造函数" << endl;
    }
    CTestA(double a, double b) // 带参的构造函数
    {   x = a;
        y = b;
        cout << "调用带参构造函数" << endl;
    }
    void dispxy(){
        cout << "x=" << x << "," << "y=" << y << endl << endl;
    }
private:
    double x, y;
};
int main()
{
    CTestA* pobja, * pobjb; // 创建两个对象指针
    pobja = new CTestA;  // 用new动态开辟存储空间，调用缺省构造函数
    pobjb = new CTestA(200, 800); // 用new动态开辟对象存储空间，调用带参构造函数
    pobja->dispxy();
    pobjb->dispxy();
    delete pobja; // 用delete释放动态开辟存储空间
    delete pobjb; // 用delete释放动态开辟存储空间
    return 0;
}
/*
```

```
调用缺省构造函数
调用带参构造函数
x=0,y=0
x=200,y=800
*/
```

析构函数

析构函数的作用与构造函数正好相反，**是在对象的生命期结束时，释放系统为对象所分配的空间，即要撤消一个对象。**

析构函数也是类的成员函数，定义析构函数的格式为：

```
ClassName::~ClassName( )
    {
        ......
        //  函数体;
    }
```

【析构函数的特点】

1、析构函数是成员函数，函数体可写在类体内，也可写在类体外。

2、析构函数是一个特殊的成员函数，函数名必须与类名相同，并在其前面加上字符“~”，以便和构造函数名相区别。

3、析构函数**不能带有任何参数，不能有返回值，不指定函数类型。**

4、一个类中，只能定义一个析构函数，**析构函数不允许重载。**

5、析构函数是在撤消对象时由系统自动调用的。

在程序的执行过程中，当遇到某一对象的生存期结束时，系统自动调用析构函数，然后再收回为对象分配的存储空间。

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

class CTestA
{
public:
    CTestA()
    {
        cout << "调用缺省的构造函数." << endl;
    }
    CTestA(double a, double b)
    {
        x = a;
        y = b;
        cout << "调用带参的构造函数." << endl;
    }
    ~CTestA()//析构函数
    {
        cout << "调用析构函数." << endl;
```

```
    }
private:
    double x, y;
};
int main()
{
    CTestA obj1;
    CTestA obj2(5.6, 70.8);
    CTestA boj3(10, 20);
    return 0;
}
```

在程序的执行过程中，对象如果用new运算符开辟了空间，则在类中**应该定义一个析构函数**，并在析构函数中使用delete删除由new分配的内存空间。**因为在撤消对象时，系统自动收回为对象所分配的存储空间，而不能自动收回由new分配的动态存储空间。**

【缺省的析构函数】

若在类的定义中没有显式地定义析构函数时，则编译器自动地产生一个缺省的析构函数，其格式为：

```
ClassName::~ClassName() { };
```

任何对象都必须有构造函数和析构函数，但在撤消对象时，**要释放对象的数据成员用new运算符分配的动态空间时，必须显式地定义析构函数。**

```
#include <iostream>
using namespace std;
class CTestA
{
public:
    CTestA()
    {
        cout << "\n调用构造函数CTestA::CTestA().\n";
        pc = new int(10000);//pc指针开辟一个int类型空间，空间赋值10000
        cout << *pc << endl;

    }
    ~CTestA()
    {
        delete pc;//释放掉pc指针开票的空间
        cout << "\n调用析构函数~CTestA::CTestA()." << endl;
    }
private:
    int* pc;
};
int main()
{
    CTestA* p1 = new CTestA;//使用 new 来分配一个 CTestA 对象。这种方式会调用 CTestA 的构造函数。
    CTestA* p2 = (CTestA*)malloc(sizeof(CTestA));//使用 malloc 分配内存，这种方式不会调用 CTestA 的构造函数，只是简单地分配了内存。
    delete p1;// 使用 delete 来释放 p1 指向的内存，这会调用 CTestA 的析构函数。
```

```
    free(p2);//使用 free 来释放 p2 指向的内存，但这不会调用 CTestA 的析构函数。
    return 0;
}
```

【实现类型转换的构造函数】

> 同类型的对象可以相互赋值，相当于类中的数据成员相互赋值；
>
> 如果**直接将数据赋给对象**，所赋入的数据需要强制类型转换，**这种转换需要调用构造函数。**

```
#include <iostream>
using namespace std;
class CTestA
{
public:
    CTestA(double a, double b)
    {
        x = a;
        y = b;
        cout << "调用带参的构造函数." << endl;
    }
    ~CTestA()
    {
        cout << "调用析构函数." << endl;
    }
    void dispxy()
    {
        cout << x << "," << y << endl << endl;
    }
private:
    double x, y;
};
int main()
{
    CTestA obj1(2, 3);
    obj1.dispxy();
    obj1 = CTestA(90, 20);//重新赋值需要重新调用构造函数，加类名
    obj1.dispxy();
    return 0;
}
```

拷贝构造函数

拷贝构造函数，又称复制构造函数，是一种特殊的构造函数，它由编译器调用来完成一些基于同一类的其他对象的构建及初始化。**其形参必须是引用，但并不限制为const，一般普遍的会加上const限制。**此函数经常用在函数调用时用户定义类型的值传递及返回。拷贝构造函数要调用基类的拷贝构造函数和成员函数。如果可以的话，它将用常量方式调用，另外，也可以用非常量方式调用。

拷贝构造函数是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。拷贝构造函数通常用于：

•通过使用另一个同类型的对象来初始化新创建的对象；

•复制对象把它作为参数传递给函数；

•复制对象，并从函数返回这个对象。

如果在类中没有定义拷贝构造函数，编译器会自行定义一个。**如果类带有指针变量，并有动态内存分配，则它必须有一个拷贝构造函数。**

拷贝构造函数的最常见形式如下：

```
classname (const classname &obj) {
   // 构造函数的主体
}
```

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

class CTestA
{
private:
    int* ptr;
public:
    CTestA(int a); // 构造函数
    CTestA(const CTestA& obj); // 拷贝构造函数
    ~CTestA(); // 析构函数

    int getptr()
    {
        return *ptr;
    }
 };

CTestA::CTestA(int a) // 构造函数
{
    cout << "调用构造函数." << endl;
    // 为指针分配内存空间
    ptr = new int;
    *ptr = a;
}

CTestA::CTestA(const CTestA& obj) // 拷贝构造函数
{
    cout << "调拷贝构造函数并为指针ptr分配内存空间.\n";
    ptr = new int;
    *ptr = *obj.ptr; // 拷贝值
}

CTestA::~CTestA() // 析构函数
{
    cout << "释放内存空间." << endl;
    delete ptr;
}

void dispptr(CTestA obj)
{
    cout << "ptr值为:" << obj.getptr() << endl;
```

```
}

int main()
{
    CTestA obj(5600);
    dispptr(obj);

    return 0;
}
```

day5

一、友元

友元是一种定义在类外部的普通函数或类，但它需要在类体内进行说明，为了与该类的成员函数加以区别，在说明时前面加以关键字friend。友元不是成员函数，但是它可以访问类中的私有成员。

类具有封装和信息隐藏的特性。只有类的成员函数才能访问类的私有成员，程序中的其他函数是无法访问私有成员的。非成员函数可以访问类中的公有成员，但是如果将数据成员都定义为公有的，这又破坏了隐藏的特性。另外，应该看到在某些情况下，特别是在对某些成员函数多次调用时，由于参数传递，类型检查和安全性检查等都需要时间开销，而影响程序的运行效率。

目 的：提高程序的运行效率

缺 点：破坏了类的封装性和隐藏性

1、友元函数

类中私有和保护的成员在类外不能被访问。友元函数是一种定义在类外部的普通函数，其特点是能够访问类中私有成员和保护成员，即类的访问权限的限制对其不起作用。

友元函数需要在**类体内**进行说明，在前面加上关键字**friend**。

一般格式为：

```
friend   float Func(Student &a);
//关键字  返回值类型 函数名 函数参数
```

友元函数不是成员函数，用法也与普通的函数完全一致，只不过它能访问类中所有的数据。**友元函数破坏了类的封装性和隐蔽性，使得非成员函数可以访问类的私有成员。**

一个类的友元可以自由地用该类中的**所有成员**。

有关友元函数的使用，说明如下：

1）**友元函数不是类的成员函数**

2）友元函数近似于**普通的函数**，它不带有this指针，因此**必须将对象名或对象的引用作为友元函数的参数**，这样才能访问到对象的成员。

```
#include <iostream>
using namespace std;
class CTestFriend
{
public:
    CTestFriend() // 默认的构造函数
```

```
    {
        cout << "调用默认的构造函数CTestFriend::CTestFriend().\n";
    }
    CTestFriend(int a, int b) // 带有参数的构造函数
    {
        x = a;
        y = b;
        cout << "x=" << x << "," << "y=" << y << endl;
    }
    int mulxy()   // 普通成员函数
    {
        return x * y;
    }
    friend int sumxy(CTestFriend &obj);  // 友元函数 可访问私有成员和保护成员
private:
    int x, y;//x,y是CTestFriend类的私有成员
};
int sumxy(CTestFriend &obj)//友元函数的外部定义
{
    return obj.x + obj.y;//访问类的私有成员，破坏了类的封装性和隐蔽性
}
int main()
{
    CTestFriend obja;//声明obja调用默认构造函数
    CTestFriend objb(50, 100);//声明objb调用传参的构造函数
    cout << "\n两个数字之积为:" << objb.mulxy() << endl << endl;//普通成员函数访问私有成员
    cout << "\n两个数字之和为:" << sumxy(objb) << endl << endl;//使用友元函数也能访问私有成员
    return 0;
}
```

2、友元函数与一般函数的不同点在于：

1）友元函数**必须在类的定义中说明**，其函数体可在类内定义，也可在类外定义；

2）它可以访问该类中的**所有成员（公有的、私有的和保护的）**，而一般函数只能访问类中的公有成员。

友元函数不受类中访问权限关键字的限制，可以把它放在类的私有部分，放在类的公有部分或放在类的保护部分，其作用都是一样的。换言之，**在类中对友元函数指定访问权限是不起作用的。**

友元函数的作用域与一般函数的作用域相同。**谨慎使用友元函数。**通常使用友元函数来**取**对象中的数据成员值，而**不修改**对象中的成员值，则肯定是安全的。

3、友元类

友元除了函数以外，还可以是类，即一个类可以作另一个类的友元。当一个类作为另一个类的友元时，这就意味着**这个类的所有成员函数都是另一个类的友元函数，都可以访问**另一个类中的隐藏信息（包括私有成员和保护成员）。

定义友元类的语句格式如下：

```
friend class 类名（即友元类的类名）;
```

注意事项

1)友元关系不能被继承。

2)友元关系是单向的，不具有交换性。若类B是类A的友元，类A不一定是类B的友元，要看在类中是否有相应的声明。

3)友元关系不具有传递性。若类B是类A的友元，类C是B的友元，类C不一定是类A的友元，同样要看类中是否有相应的申明。

```
#include <iostream>
using namespace std;
class CTestFClassA
{
public:
    CTestFClassA(double a, double b) {
        x = a;
        y = b;
    }
    double getx() {
        return x;
    }
    double gety() {
        return y;
    }
    friend class CTestFClassB; // 定义类CTestFClassB为类CTestFClassA友元
private:
    double x, y;
};
class CTestFClassB
{
public:
    CTestFClassB(int n = 1)
    {
        k = n;
    }
    void disp(CTestFClassA &obj) {
        cout <<"结果为:"<< obj.x + obj.y + k;//可以访问到类CTestFClassA中的私有成员x,y
    } // 求类CTestFClassA的某个数据成员到这边来操作。
private:
    int k;
};
int main()
{
    CTestFClassA obj1(1, 2), obj2(3, 4);
    CTestFClassB objb(5);
    objb.disp(obj1);
    cout << endl;
    objb.disp(obj2);
    return 0;
}
```

二、动态内存new/delete

在定义变量或数组的同时即**在内存为其开辟了指定的固定空间。**

```
int n,  a[10];

char  str[100];
```

一经定义，即为固定地址的空间，在内存不能被别的变量所占用。

在程序内我们有时需要根据实际需要开辟空间，如输入学生成绩，但每个班的学生人数不同，一般将人数定得很大，这样占用内存。

```
#define   N    1000
......
float    score[N][5];
cin>>n;
for(int i=0;i<n;i++)
    for(j=0;j<5;j++)
        cin>>score[i][j];
......
/*
无论班级中有多少个学生，程序均在内存中开辟1000×5个实型数空间存放学生成绩，造成内存空间的浪费。
*/
```

利用 new 运算符可以在程序中动态开辟内存空间。

```
new  数据类型[单位数];
new int[4];
```

在内存中开辟了4个int型的数据空间，即16个字节

new 相当于一个函数，在内存开辟完空间后，返回这个空间的首地址，这时，**这个地址必须用一个指针保存下来**，才不会丢失。

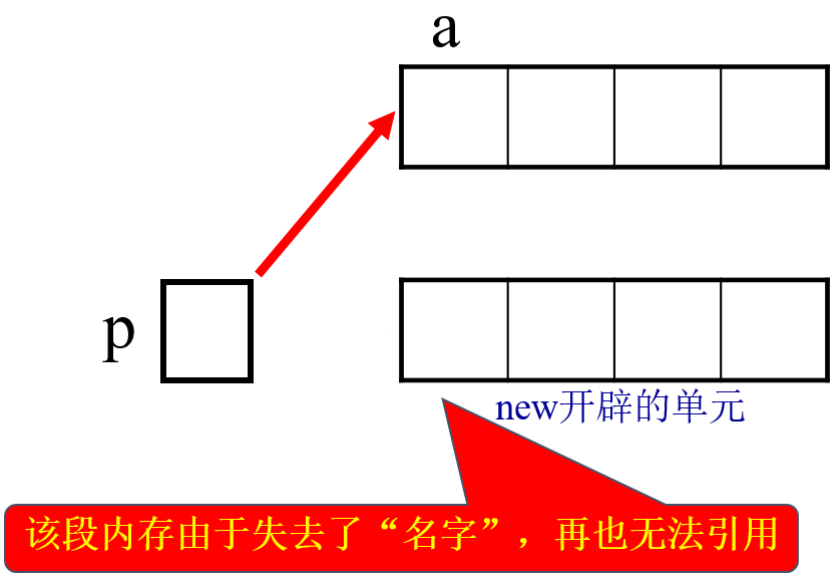
```
int* p;//声明一个int类型的指针
p = new int;//用指针接收new开辟的整形空间的首地址
*p = 6;//把6放到这个空间里
//可以用*p对这个空间进行运算。
```

同样，利用new运算符也可以开辟连续的多个空间(数组)。

```
int  n,* p;
cin>>n;
p=new int[n];
//p指向新开辟空间的首地址
for(int i=0;i<n;i++)
    cin>>p[i];
//可以用p[i]的形式来引用新开辟的内存单元。
```

注意：用new开辟的内存单元没有名字，指向其首地址的指针是引用其的唯一途径，若指针变量重新赋值，则用new开辟的内存单元就在内存中“丢失”了，别的程序也不能占用这段单元，直到重新开机为止。

```
int  * p, a[4];
p=new int[4];
p=a;
```



用 new 运算符分配的空间，**不能在分配空间时进行初始化。**

同样，用new开辟的内存单元如果程序不“主动”收回，那么这段空间就一直存在，直到重新开机为止。

delete运算符用来将动态分配到的内存空间归还给系统，使用格式为：

```
delete p;
```

```
int  *point;
point=new  int;
......//注意：在此期间，point指针不能重新赋值
delete   point;//只有用new开辟的空间才能用delete收回
//delete也可以收回用new开辟的连续的空间
int  *point;
cin>>n;
point=new int[n];
.......//当内存中没有足够的空间给予分配时，new 运算符返回空指针NULL（0）
delete  [ ]point;
```

```
#include <iostream>
using namespace std;
class CNewDelete
{
public:
    CNewDelete() {
        cout << "\n调用默认的构造函数" << endl;
        str = new char[10];
    }
    ~CNewDelete() {
        cout << "\n调用析构函数" << endl;
        delete[]str;
    }
private:
```

```
    char *str;
};
int main()
{
    char buffer[100]; //栈上分配缓冲区
    CNewDelete *pobj = new(buffer) CNewDelete();//这是对 CNewDelete 类的默认构造函数的调用。它在 buffer 指向的内存区域中构造一个 CNewDelete 对象
    //new(buffer) 是在已分配的内存区域 buffer 中构造对象的语法
    pobj->~CNewDelete();  // 主动调用析构函数，避免内存泄漏
    char *buffer2 = new char[100];//堆上分配缓冲区
    CNewDelete *pobj2 = new(buffer2) CNewDelete();
    pobj2->~CNewDelete(); // 必须记住，主动调用析构函数
    delete[]buffer2; // 堆内存需要主动释放操作
    return 0;
}
```

三、运算符重载

可以重定义或重载大部分 C++ 内置的运算符。这样，您就能使用自定义类型的运算符。

重载的运算符是带有特殊名称的函数，函数名是由关键字 operator 和其后要重载的运算符符号构成的。与其他函数一样，重载运算符有一个返回类型和一个参数列表。

```
Box operator+(const Box&);
```

声明加法运算符用于把两个 Box 对象相加，返回最终的 Box 对象。大多数的重载运算符可被定义为普通的非成员函数或者被定义为类成员函数。如果我们定义上面的函数为类的非成员函数，那么我们需要为每次操作传递两个参数，如下所示：

```
Box operator+(const Box&, const Box&);
```

在C++中，允许重载的运算符如下：

运算符	详情
双目算数运算符	+，-，*，/，%
关系运算符	==，!=，<，>，<=，>=
逻辑运算符	||，&&，！
单目运算符	+(正)，-(负)，*(指针)，&(取地址)
自增自减运算符	++，--
位运算符	|，&，~，^，<<，>>
赋值运算符	=，+=，-=，*=，/=，%=，&=，|=，^=，<<=，>>=
空间申请与释放	new，delete，new[]，delete[]
其他运算符	()(函数调用)，->(成员访问)，，(逗号)，

在C++中，不允许重载的运算符如下：

```
. 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

\# 预处理符号
```

1、重载运算符和重载函数

一元运算符只对一个操作数进行操作，下面是一元运算符的实例：

递增运算符（++）和递减运算符（--）

一元减运算符，即负号（-）

逻辑非运算符（!）

一元运算符通常出现在它们所操作的对象的左边，比如 !obj、-obj 和 ++obj，但有时它们也可以作为后缀，比如 **obj++ 或 obj--**。

2、二元运算符重载

二元运算符需要两个参数，下面是二元运算符的实例。我们平常使用的加运算符（+）、减运算符（-）、乘运算符（*）和除运算符（/）都属于二元运算符。就像加(+)运算符。

```
#include <iostream>
using namespace std;
class Box
{
public:
    Box()
    {
        l = 0;
        b = 0;
        h = 0;
    }
    void setl(double length)
    {
        l = length;
    }
    void setb(double breadth)
    {
        b = breadth;
    }
    void seth(double hight)
    {
        h = hight;
    }
    double getv(void)  // 求体积
    {
        return l * b * h;
    }
    // 重载 + 运算符，用于把两个Box对象进行相加
    Box operator+(const Box& b)
    {
```

```
        Box box;
        box.l = this->l + b.l;
        box.b = this->b + b.b;
        box.h = this->h + b.h;
        return box;
    }
private:
    double l; // 长度
    double b; // 宽度
    double h; // 高度
};
int main()
{
    Box box1; // 声明box1，类型为Box
    Box box2;
    Box box3;
    double dv = 0.0; // 存储体积变量
    // 为第一个对象赋值 box1
    box1.setl(2.0);
    box1.setb(3.0);
    box1.seth(4.0);
    // 为第二个对象赋值 box1
    box2.setl(5.0);
    box2.setb(6.0);
    box2.seth(7.0);
    // 输出box1体积
    dv = box1.getv();
    cout << "box1对象的体积为:" << dv << endl << endl;
    // 输出box2体积
    dv = box2.getv();
    cout << "box2对象的体积为:" << dv << endl << endl;
    // 把两个对象相加，得到 box3
    box3 = box1 + box2;
    // 输出obx3体积
    dv = box3.getv();
    cout << "box3对象的体积为:" << dv << endl << endl;
    return 0;
}
```

3、++ 和 -- 运算符重载

递增运算符（++）和递减运算符（--）是 C++ 语言中两个重要的一元运算符。

```
#include <iostream>
using namespace std;
class CTestSS
{
public:
    CTestSS() :i(0) {

    }
    CTestSS operator++()
    {
        cout << "调用前置递增.\n";
        CTestSS obj;
        obj.i = ++i;
```

```
        return obj;
    }
    // 括号中插入int 表示后缀
    CTestSS operator++(int)
    {
        cout << "调用后置递增.\n";
        CTestSS obj;
        obj.i = i++;
        return obj;
    }
    void disp()
    {
        cout << "i=" << i << endl ;
    }
private:
    int i;
};
int main()
{
    CTestSS obj1, obj2;
    cout << "\n\n输出结果1为:" << endl;
    obj1.disp();
    obj2.disp();
    // 调用运算符函数，然后将obj1的值赋给obj2
    obj2 = ++obj1;
    cout << "\n\n输出结果2为:" << endl;
    obj1.disp();
    obj2.disp();
    obj2 = obj1++;
    cout << "\n\n输出结果3为:" << endl;
    obj1.disp();
    obj2.disp();
    return 0;
}
```

```
#include <iostream>
using namespace std;
class CTest
{
public:
    CTest() :x(10) {

    }
    CTest operator--()
    {
        CTest obj;
        obj.x = --x;
        return obj;
    }
    // 括号中插入int参数，表示后缀计算方式
    CTest operator--(int)
    {
        CTest obj;
        obj.x = x--;
        return obj;
```

```
    }
    void Disp()
    {
        cout << "x=" << x << endl;
    }
private:
    int x;
};
int main()
{
    CTest obj1, obj2;
    cout << "\n\n输出结果1为:" << endl;
    obj1.Disp();
    obj2.Disp();
    obj2 = --obj1;
    cout << "\n\n输出结果2为:" << endl;
    obj1.Disp();
    obj2.Disp();
    obj2 = obj1--;
    cout << "\n\n输出结果3为:" << endl;
    obj1.Disp();
    obj2.Disp();
    return 0;
}
```