1. cout << 数据

数据的值默认十进制显示

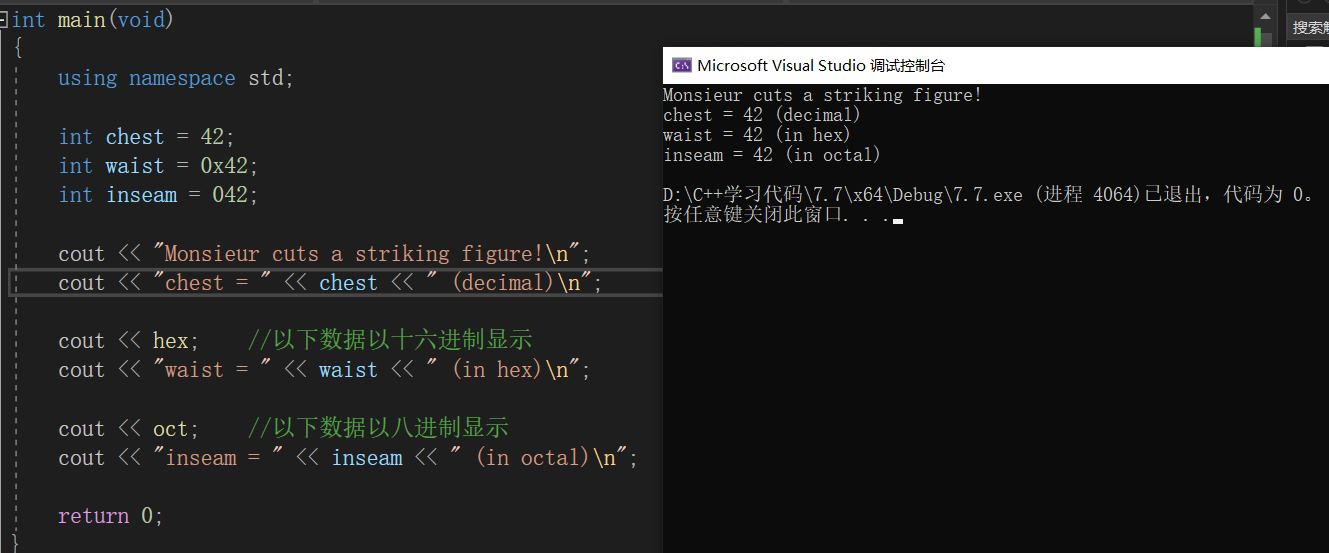
cout << hex (十六进制)

该语句下面数据以十六进制打印数据的值

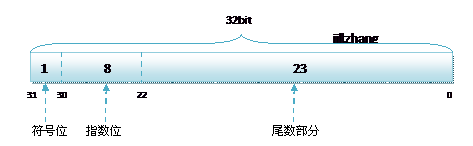
cout << oct(八进制)

该语句下面的数据以八进制打印数据的值

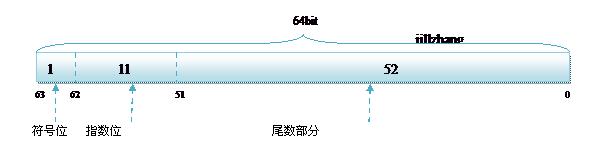
代码案例：



1. 和整形一样，浮点数在计算中的存储也是二进制存储，因此内存中要存储一个浮点数，首先要将该数的整数部分和小数部分转化成二进制数，如8.25，转换成二进制数1000.01，再用2进制科学计数法表示为1.00001 \* 23。



float类型



double类型

符号位存储形式（占1位）：整数则为0，负数则为1。例如1.00001 \* 23是正数，因此符号位为0

指数位存储形式（占8位或11位）：通过将真实的指数位进行偏移之后存储，所以指数位可以存储在区间[-(2^(n-1)-1), 2^n]之间的数据，大于这个区间则数据会溢出.所以使用32bit进行存储时,n=8,指数位真实值需要+127得到真实的指数位存储值；使用64位进行存储时，n=11,指数位真实值+1023得到指数位的存储值。例如1.00001 \* 23的指数位是3，以32bit存储，指数位的值是127 + 3 = 130，再转化为二进制数则为10000010。

尾数部分存储形式（占剩余位数）：1.00001 \* 23尾数部分是00001，因此在内存中存储的就是00001。

在内存中的存储：符号位：0；指数位10000010；尾数：00001。

若为32bit存储，则8.25在内存中存储形式是：

0 10000010 00001 000000000000000000

符号 指数 尾数 补足32位

C++对于有效位的要求是，float至少32位，double至少48位。

float的有效位数：float的32位有效数字是对2进制数来说。每4位2进制数表示1位十进制数，即32位的2进制数就是8位的10进制数。float至少有6位有效位，同理，浮点数的符号位和指数位总共占了9位，因此小数部分占23位，在加上整数位1位，总共24位，24 ÷ 4 = 6，因此，浮点数的十进制有效位是6位。

float和double的精度：float的小数部分的有效位是23，即十进制的6位有效，而double的小数部分的有效位是52，即十进制的13位有效

float和double在内存中存储的是一个近似值（精度降低）：例如11.17，这个数若以2进制表示则其小数位是无限的，而对于float的小数位有效位是23位（10进制是6位），因此超过了这个23位（十进制是6位）则数据是不准确的；double类型小数位的有效位则是52位（十进制13位），超过了52位也是不准确的数。

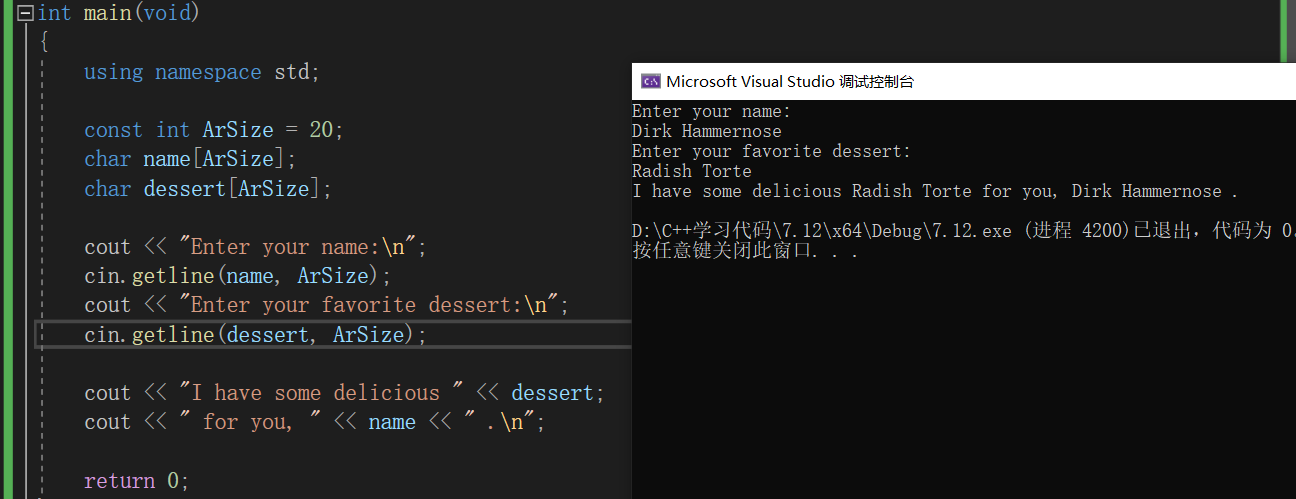
1. **字符串和字符数组**

区别:字符串和字符数组都可以存在字符数组中，以\0为结尾的字符存储在字符数组中称为字符串，不以\0结尾的字符存储在字符数组中称为字符数组。

1. **面向行的输入：getline()函数**

getline（）函数读取整行，它通过回车键输入的换行符来确定输入结尾。要调用这种方法，可以使用cin.getline()。该函数有两个参数。第一个参数是用来存储输入行的数组的名称，第二个参数是要读取的字符数。如果这个参数为20，则函数最多读取19个字符，余下的空间用于存储自动在结尾处添加的空字符。getline（）成员函数读取指定数目的字符或遇到换行符时停止读取。

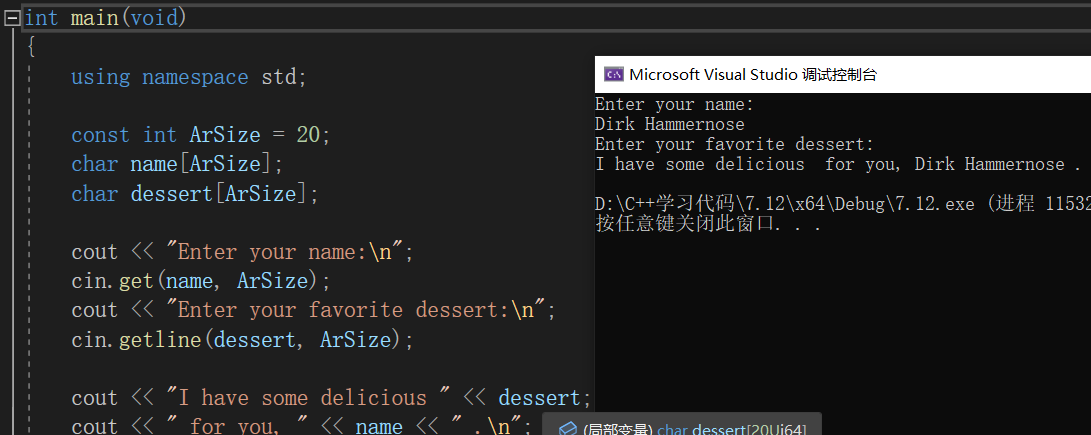
**代码例子：**



1. **面向行的输入：get()**

get()的成员函数有几种变体。其中一种变体的工作方式与getline()类似，它们接受的参数相同，解释参数的方式也相同，并且都读取到行尾。但get()并不会读取并丢弃换行符，而是将其留在输入队列中。

代码例子：



第一次调用cin.get()函数后，换行符留在输入队列中，因此第二次调用时看到的第一个字符便是换行符。因此get()认为已经到达行尾，而没有发现任何可读的内容。如果不借助于帮助，get()将不能跨过该换行符。

解决此问题的方法：

使用get()的另一种变体。使用不带任何参数的cin.get()调用可读取下一个字符(即使是换行符)，因此可以用它来处理换行符，为读取下一行输入做好准备。

用法例子：

cin.get(name, ArSize); //read the first line

cin.get(); //read newline

cin.get(name, ArSize); //read second line

cin.get(name, ArSize).get();

能使用这种方法的原因是，由于cin.get(name, ArSize)返回一个cin对象，该对象随后将被用来调用get()函数。

使用getline()和get()的区别：

假设用get()函数将一行读入数组中。如何知道停止的原因是由于已经读取了整行，而不是由于数组已经填满呢？查看下一个输入字符，如果是换行符，说明已经读取了整行；否则，说明该行中还有其他输入。

1. **string类**

可以使用string类型的变量(使用C++的说法是对象)而不是字符数组来存储字符串。string类定义隐藏了字符串的数组性质，能够像处理普通变量那样处理字符串。P69

代码案例：



使用string对象的方式与使用字符数组相同。

\*可以使用C风格字符串来初始化string对象

例如：string str2 = “panther”;

\*可以使用cin来将键盘输入存储到string对象中

例如：cin >> str1;

\*可以使用cout来显示string对象

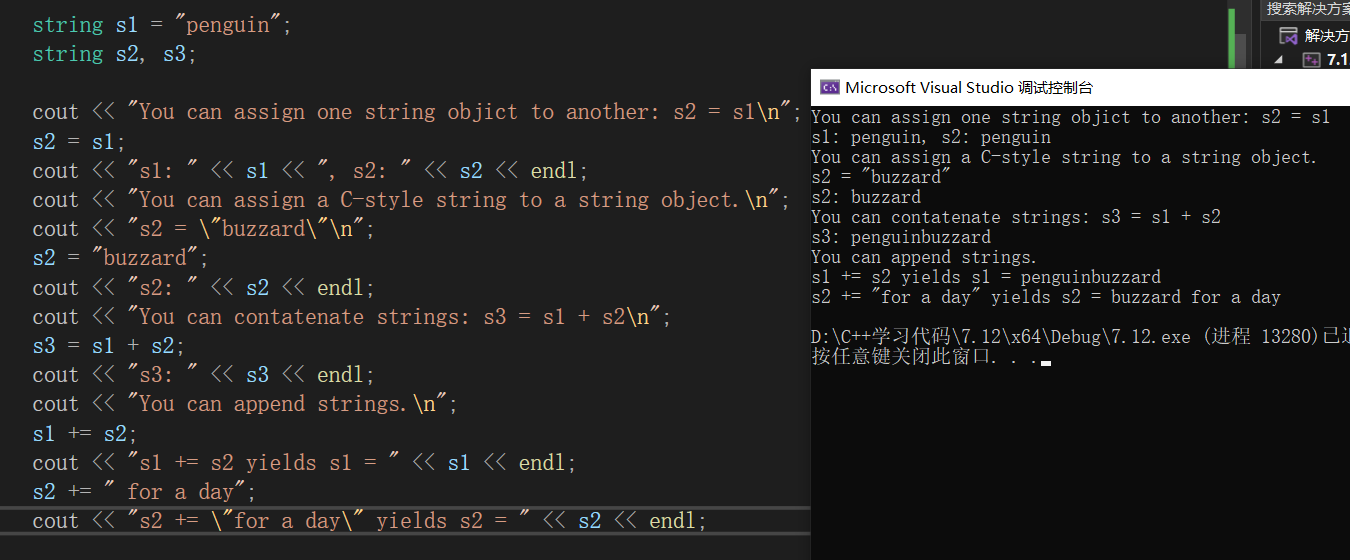
例如：cout << str1;

\*可以使用数组表示法来访问存储在string对象中的字符

例如：cout << str2[2];

赋值、拼接和附加

代码案例：



赋值

s2 = s1;

拼接

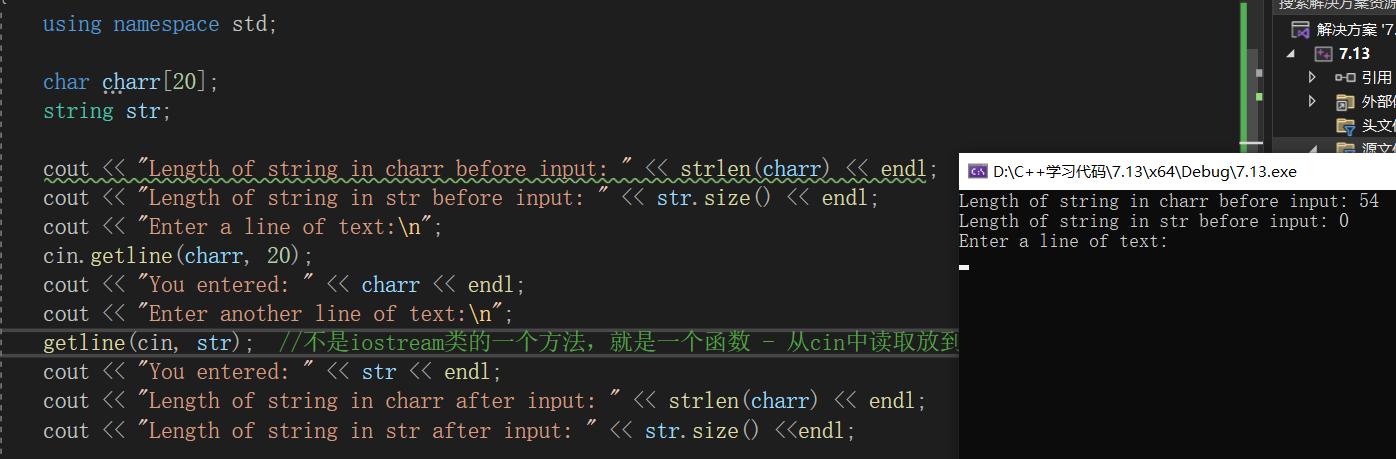
Str3 = str1 + str2； //将s1和s2加到s3

附加

str1 += str2； //add str2 to the end of str1

1. **函数strlen()和C++的size()方法**

代码案例：



在本案例中，数组的内容未初始化，因此内容是未定义的；其次，函数strlen()从数组的第一个元素开始计算字节数，直到遇到空字符(\0)，而在本案例中，直到第54个才遇到空字符。而size()方法中，str中的字符串为0，这是因为未被初始化的string对象的长度被自动设置为0。

1. **联合体（共用体）**

联合体是一种数据格式，它能够存储不同的数据类型，但同时只能同时存储其中的一个类型。

联合体的长度为其最大成员的长度。例如：

union one4all

{

char a；

int b；

}；

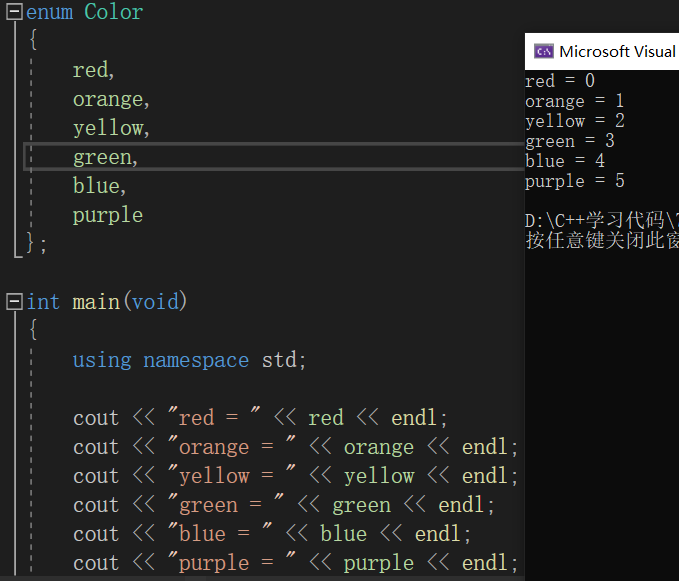
其中int类型比char类型的大，因此联合体的长度是int类型的4。

联合体的用途：当数据项使用两种或更多种格式（但不会同时使用）时，可节省空间。

1. **枚举**

枚举也是一种创建符号常量的方式，这种方式可以代替const。

代码案例：



在要定义多个字符常量时，这种创建字符常量的方法比使用const的方法更加方便。

枚举的特殊属性：

1. 对于枚举，只定义了赋值运算符，并没有为枚举定义算术运算。

例如：

band = orange； 合法

++band； 不合法

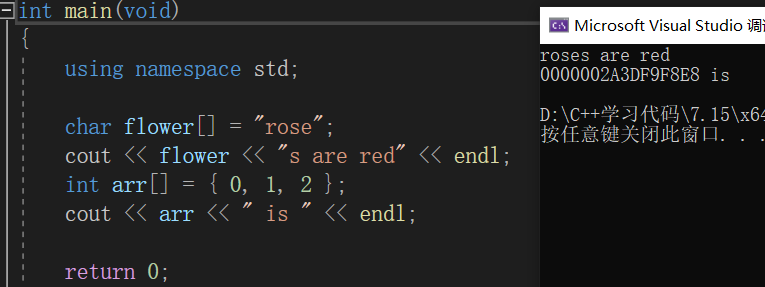
band = orange + red 不合法

1. 枚举量是整形，可以被提升为int类型，但int类型不能自动转换成枚举类型。
2. 枚举的取值范围

定义:首选，要找出上线，需要知道枚举量的最大值。找到这个最大值的、最小的2的幂。例如枚举的最大值是101，比这个数大的最小值为128，那么这个值的上限就是27 – 1 = 127；要计算下限，需要知道枚举量的最小值，如果它不小于0，则取值范围的下限为0；否则，采用与寻找上限方式相同的方式。例如枚举的最小值是-6，比这个数小的最大值是-8，即下限为2-3 + 1 = -7。

1. **cout对象认为char的地址是字符串的地址，因此它打印改地址处的字符，然后继续打印后面的字符，直到遇到空字符（\0）为止；若是整形，则将打印其地址。**

代码案例：



1. **模板类vector(动态数组)** – 数组地址在堆区

模板类类似于string类，也是一种动态数组。基本上，它是使用new创建动态数组的替代品。

要使用vector类，必须包含头文件vector。其次，vector包含在命名空间std中。

一般而言，下面的声明创建一个名为vt的vector对象，它可存储n\_elem个类型为typeName的元素：

Vector<typeName> vt(n\_elem);

例如：#include <vector>

…

using namespace std;

vector<int> vi; //创建一个大小为零的 int 数组

int n;

cin >> n;

vector<double> vd(n) //创建 n 个双精度数组

其中参数n\_elem可以是整形常量，也可以是整形变量。

1. **模板类array（静态数组）** - 数组地址在栈区

vector类的功能强大，但付出的代价是效率稍低。如果需要的是长度固定的数组，使用数组是最佳的选择。Array也位于名称空间std中，其效率与数组相同，但更方便，更安全。要创建array对象，需要包含头文件array。

下面的声明创建一个名为arr的array对象，它包含n\_elem个类型为typename的元素：

array<typename, n\_elem> arr;

例如： #include <array>

…

using namespace std;

array<int, 5> ai; //创建 5 个整数的数组对象

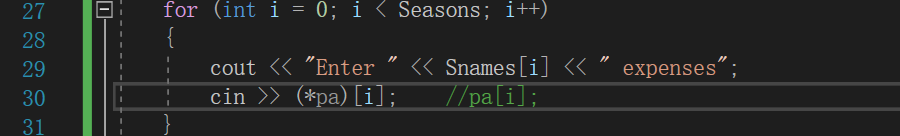
array<double, 4> ad = {1.2, 2.1, 3.43, 4.3};

与创建vector对象不同的是，n\_elem不能是变量。

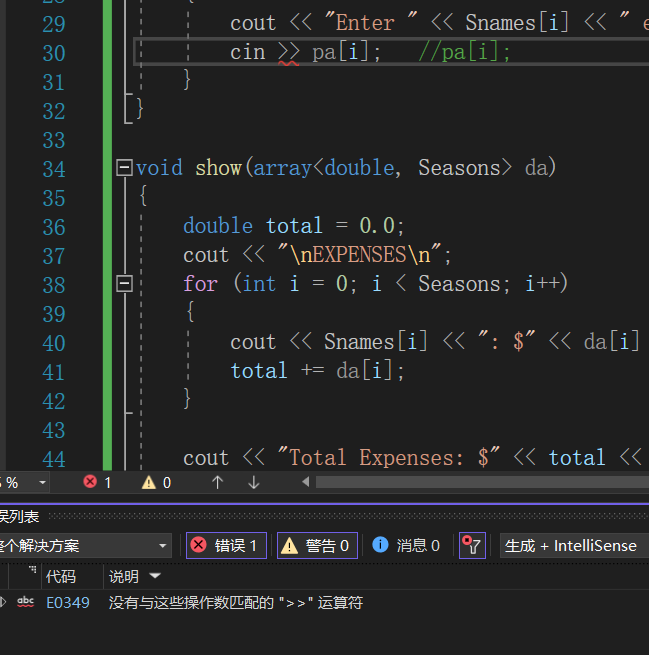
array与数组的区别：

array对象与数组的使用方式相同，但array指针不同。

代码案例：

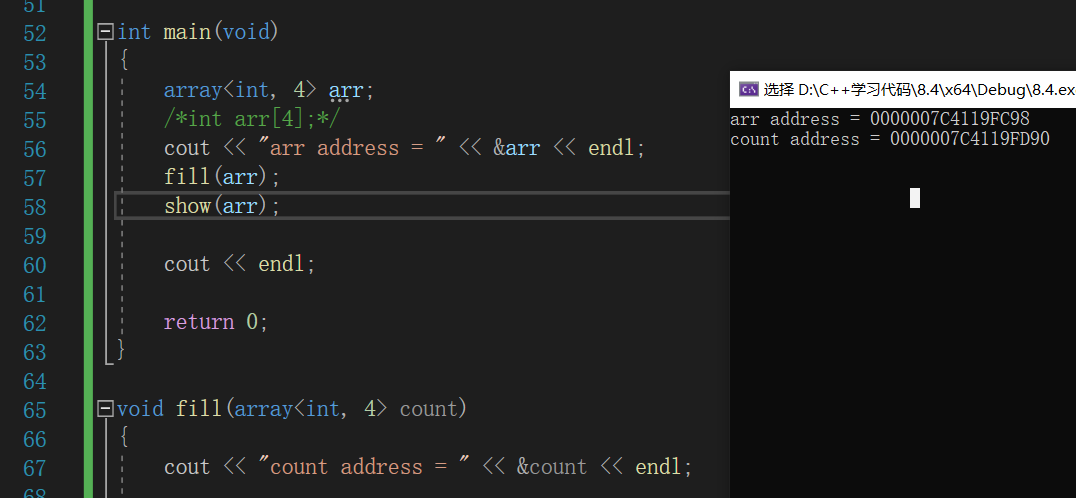


在代码第30行中，数组指针可以使用注释的用法，但array指针不能使用注释里的用法。使用会造成以下错误：



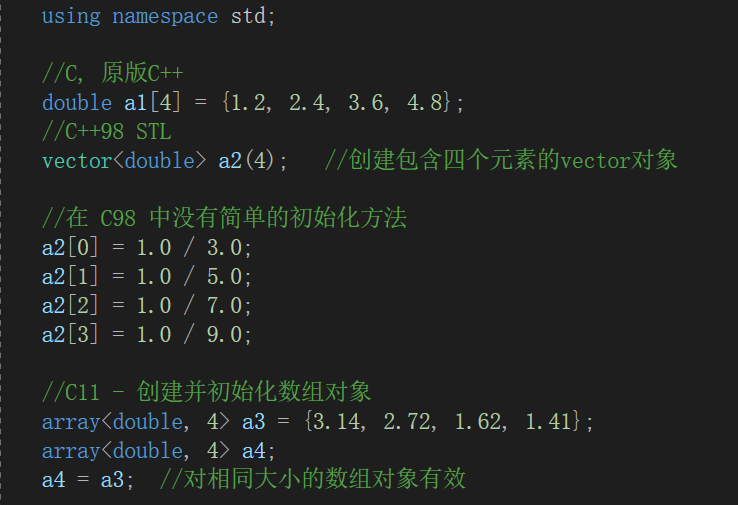
array作为参数传递时传递的是形参

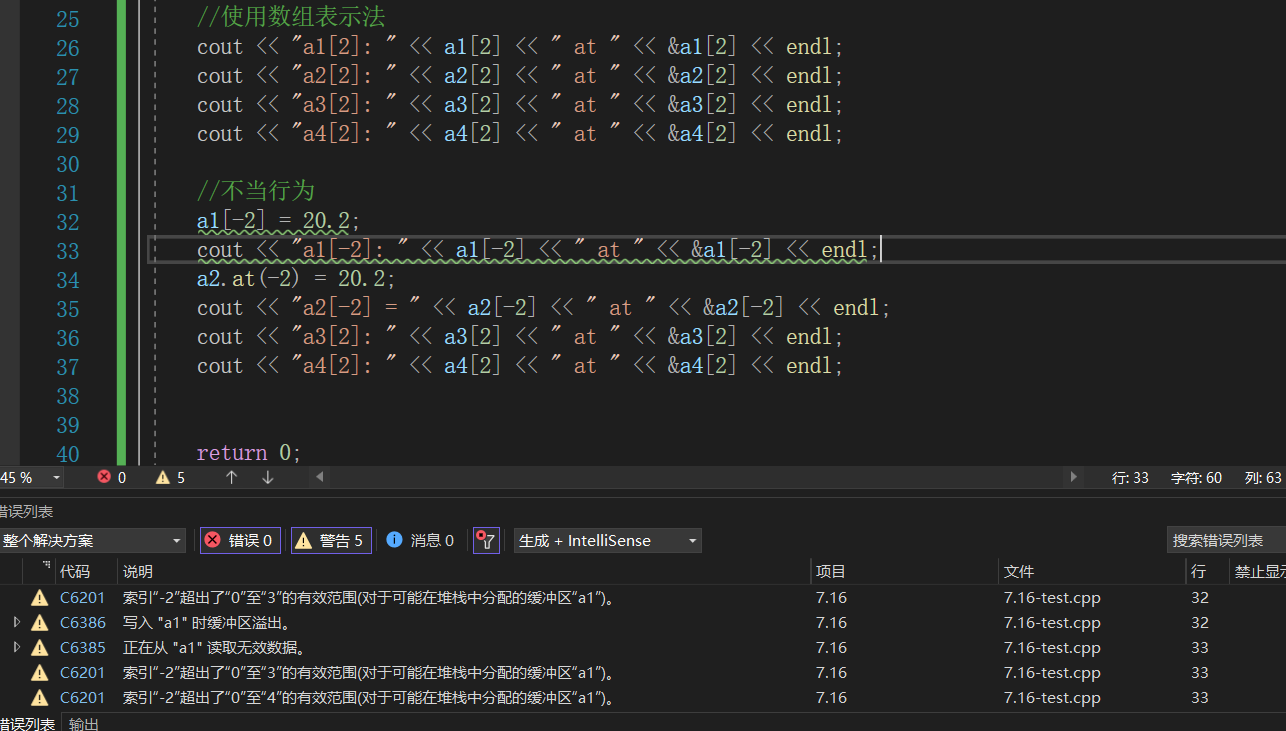
代码案例：



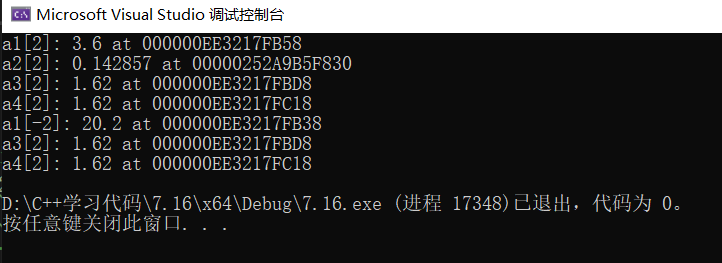
可以看到，主函数中的地址和函数中的地址是不同的。而数组传递的则是首元素地址。

1. **vector 和 array对象的成员函数at()**



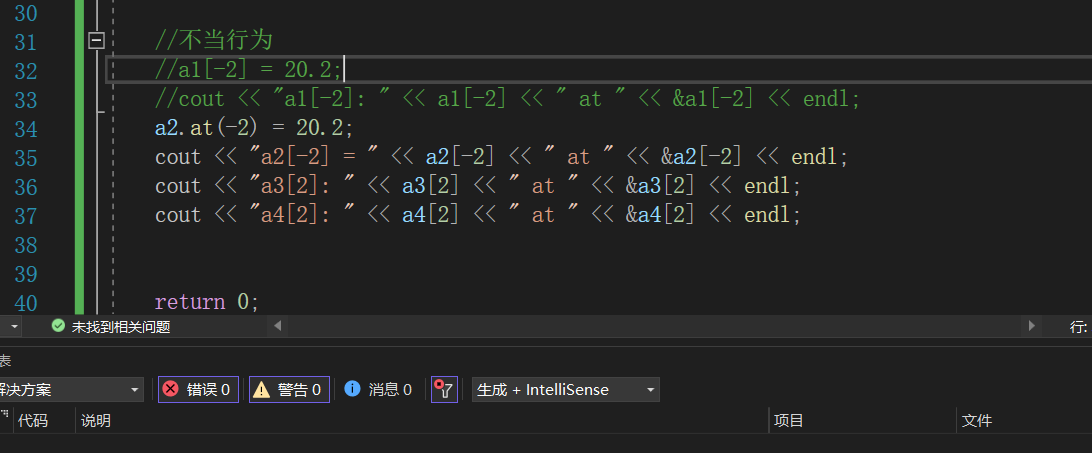


以上代码案例中，原版C数组的a1[-2] = 20.2,这里将20.2存储到了数组的外面，但C语言与C++不检查这种超界错误，其运行结果如下：



数组存在超界问题仍能正常运行，但这种数组的行为是不安全的。

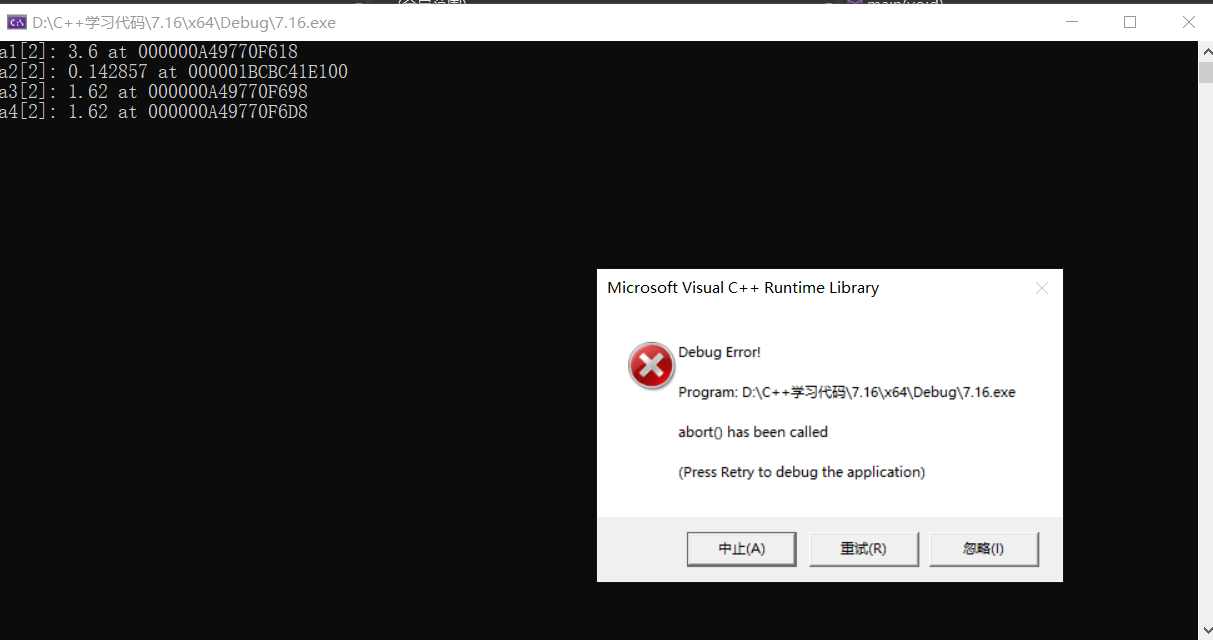
将后段不正当行为的代码改成如下形式：



vector和array对象能使用成员函数at()来禁止这样的行为：

a2.at(-2) = 2.3;

其运行结果是：

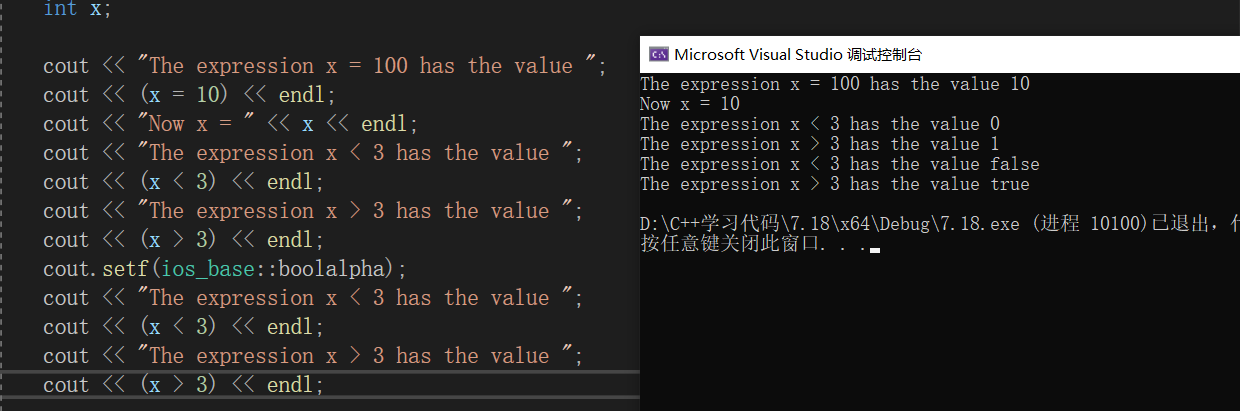


程序会中断，这就是vector和array对象的成员函数at()的作用。

1. **cout.setf(ios::boolalpha)**

该函数调用设置了一个标记，该标记命令cout显示true和false，而不是1和0.

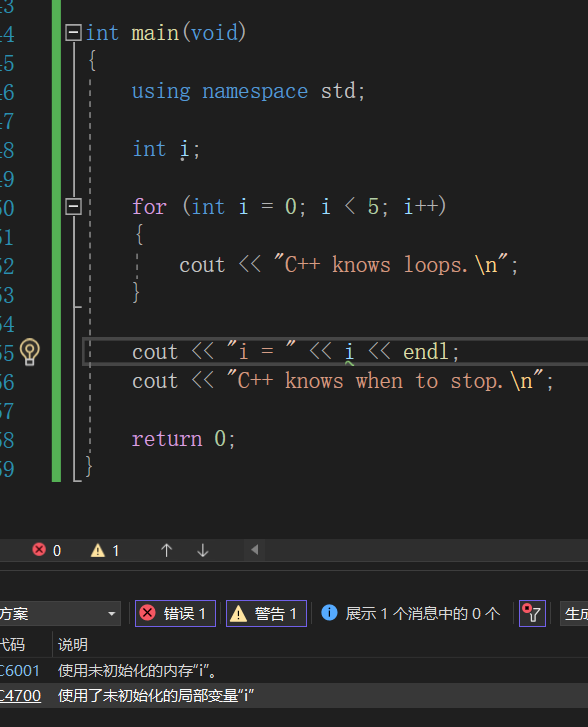
代码案例：



1. **for循环**

在for循环中，若在for循环部分中声明和初始化变量，这种变量值存在于for语句中，也就是说，当程序离开循环后，这种变量将消失。

代码案例：

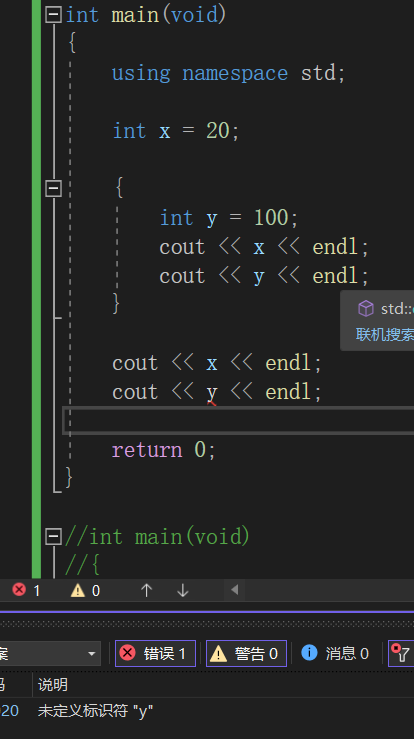


在此条代码的第10行for循环的初始化部分，对变量i进行了声明，导致i只在循环体中有效，而出了循环体则变量i无效。

**16.复合语句**

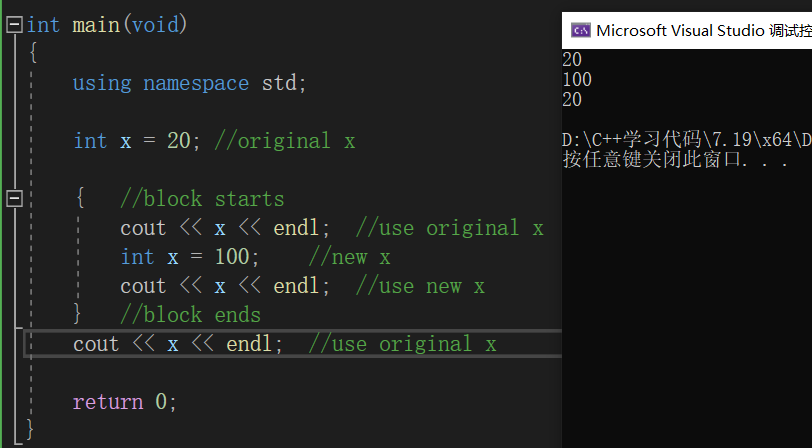
如果在语句块中定义一个新的变量，则仅当程序执行该语句中的语句时，该变量才存在。执行完该循环后，变量将被释放，表明此变量仅在该语句中才是可用的。

代码案例：



如果在一个语句块中声明一个变量，而外部语句块中也有一个这种名称的变量，则在声明位置到内部语句块结束的范围之内，新变量将隐藏旧变量，然后旧变量再次可见。

代码案例：



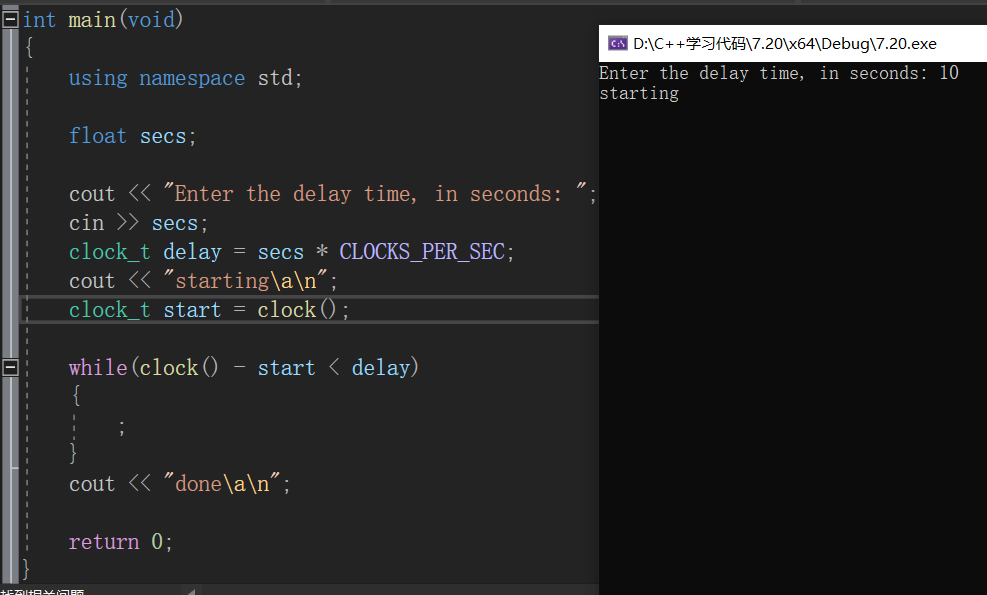
**17.clock()函数**

头文件：#include <ctime>

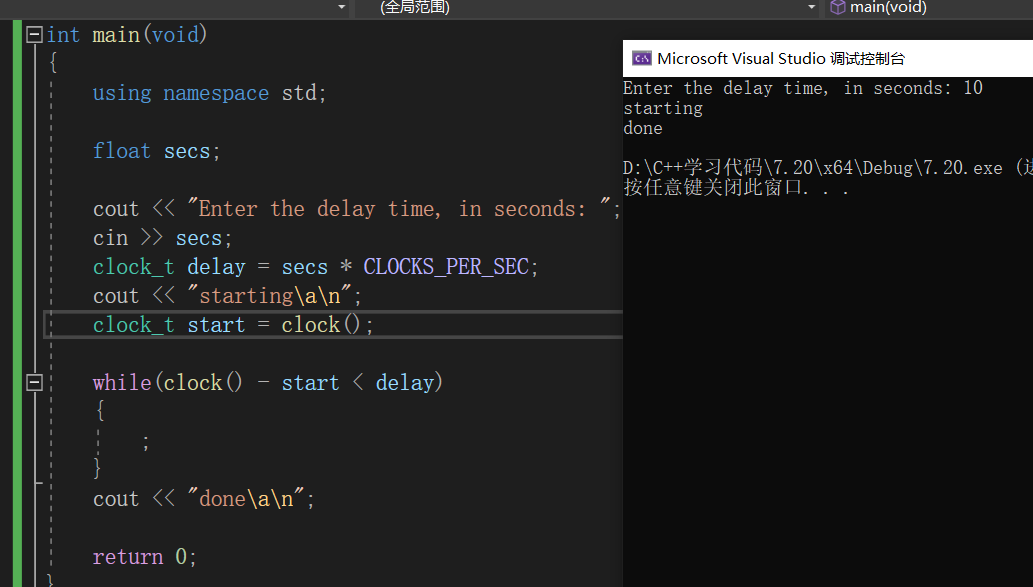
该函数返回程序开始执行后所用的系统时间。

它定义了一个符号常量----CLOCKS\_PER\_SEC（系统时间单位），该常量等于每秒钟包含的系统时间单位数。将系统时间除以这个值，可以得到秒数。或者将秒数乘以CLOCKS\_PER\_SEC，可以得到以系统时间为单位的时间。其次，ctime将clcok\_t作为clock()返回类型的别名，这意味着可以将变量声明为clock\_t类型，编译器将把它转换为long、unsigned int或适合系统的其他类型。

代码案例：



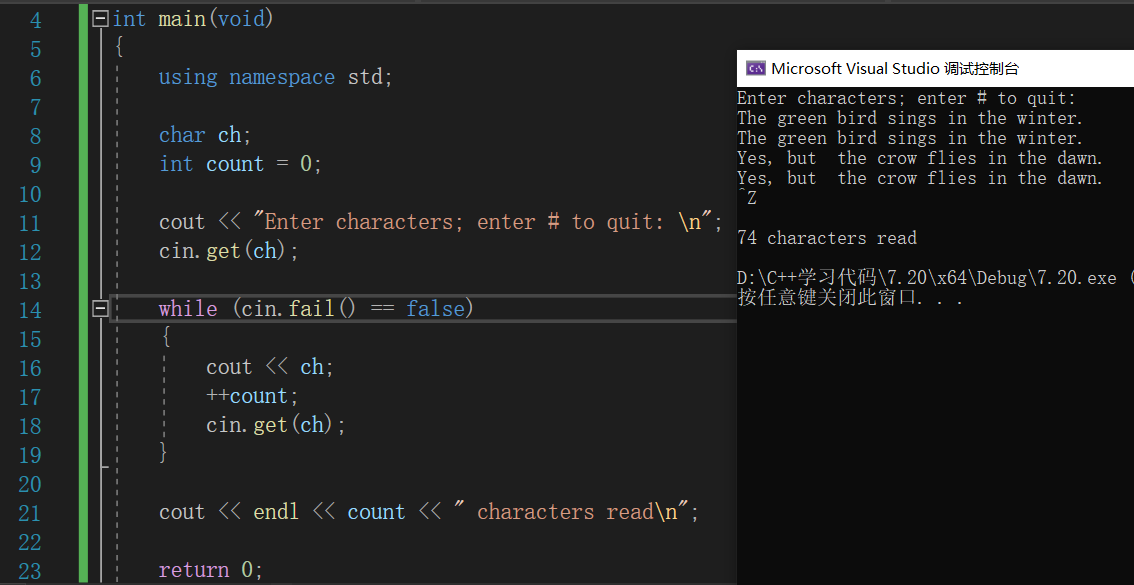
程序将等待10秒：



**18.文件尾条件**

检测到EOF后，cin将两位(eofbit和failbit)都设置为1。可以通过成员函数eof来查看eofbit是否被设置；如果检测到EOF，则cin.eof()将返回bool值true。注意，eof()和fail()方法报告最近读取的结果；也就是，它们在事后报告，而不是预先报告。因此应将cin.eof()或cin.fail测试放在读取后。

代码案例：



**19.C++中cin.get()、cout.put()和C语言中的getchar()和putchar()类型。**

**20.clear()方法**

cin方法的调用将返回false。方法返回false意味着可以用非数字输入来结束读取数字的循环。非数字输入设置错误标记意味着必须重置该标记，程序才能继续读取输入。clear()方法重置错误输入标记，同时也重置文件尾(EOF条件)。

**21.cin输入非法的情况**

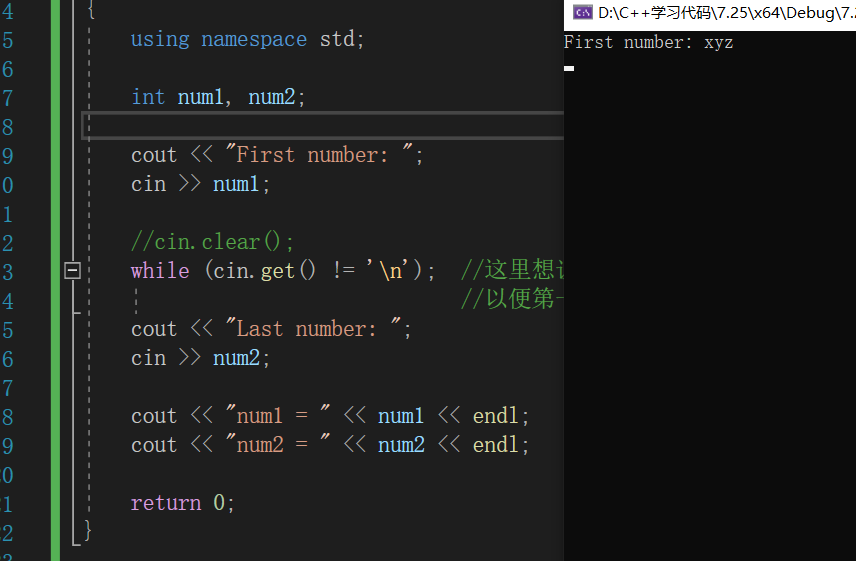
int n；

cin >> n;

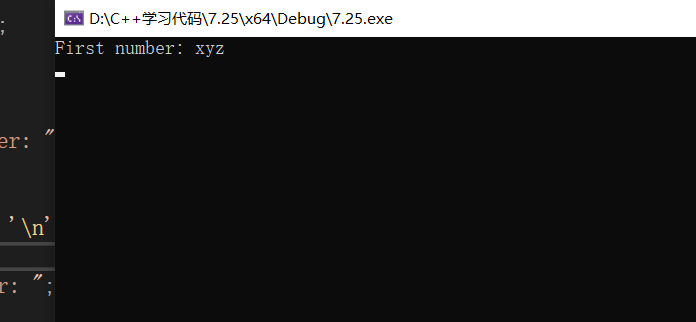
用户输入一个单词，而不是一个数字，将发生4种情况：

1. n的值保持不变
2. 不匹配的输入将被留在输入队列中
3. cin对象中的一个错误标记被设置
4. 对cin方法的调用将返回flse(如果被转换为bool类型)

代码案例：

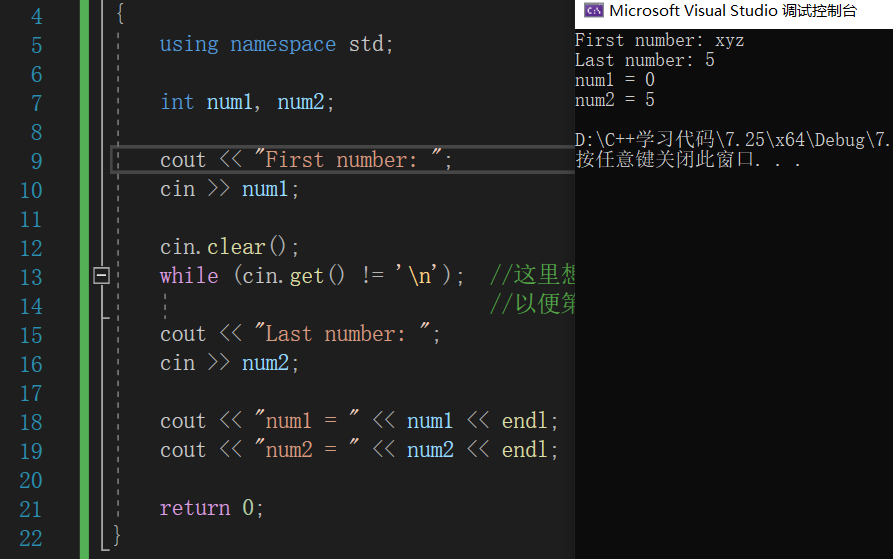


运行结果：



第一个cin非法输入导致了后面的cin的功能都失效了。

而为了解决此类问题，可使用clear()方法，此时的运行结果：

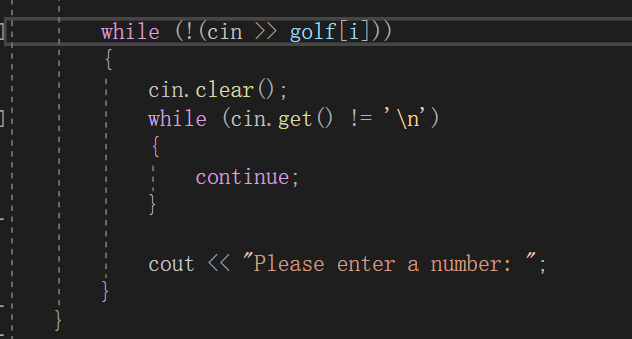


第12行使用clear()方法，num1输入非法，导致赋值失败，但while循环里的cin有了作用（将错误输入，非数字除’\n’的字符全部读取掉），第二个cin给num2赋值5成功了。

当用户输入非数字输入，程序将拒绝，并要求用户继续输入数字。可以使用cin输入表达式的值来检测输入是不是数字。程序发现用户输入了错误内容时，应采取3个步骤：

1. 重置cin以及接受新的输入
2. 删除错误输入
3. 提示用户再输入

关键代码部分：



**22.写入到文本文件**

1.包含头文件fstream;

2.声明一个或多个ofstream对象

ofstream outFile;

3.将ofstream对象与文件关联起来，使用方法之一是open()方法。

outFile.open（carinfo.txt）;

outFile的用法和cout相同：

cout << fixed; //以普通方式输出，非科学计数法

cout.precision(2); //保留2位小数点精度，括号里是几，小数点后面就保留几位

cout.setf(ios\_base::showpoint); //强制显示浮点数小数点后的0

**23.读取文本文件**

1.包含头文件ifstream

2.声明一个或多个ifstream对象

ifstream inFile;

3.将ifstream对象与文件关联其起来。使用方法之一是open()方法。

inFile.open(“bowling.txt”);

在打开文件操作，若试图打开一个不存在的文件用于输入，这种情况将导致后面使用ifstream对象进行输入时失败。检查文件是否被成功打开的首先方法是使用is\_open(),如以下类似：

inFile.open(“bowling.txt”);

if（！inFile.is\_open()）

{

exit(EXIT\_FAILURE);

}

如果文件被成功打开，方法is\_open将返回true。

24.函数exit()

函数exit()的原型是在头文件cstdlib中定义的，在该头文件中，还定义了一个用于同操作系统通信的参数值EXIT\_FAILURE。函数exit()终止程序。

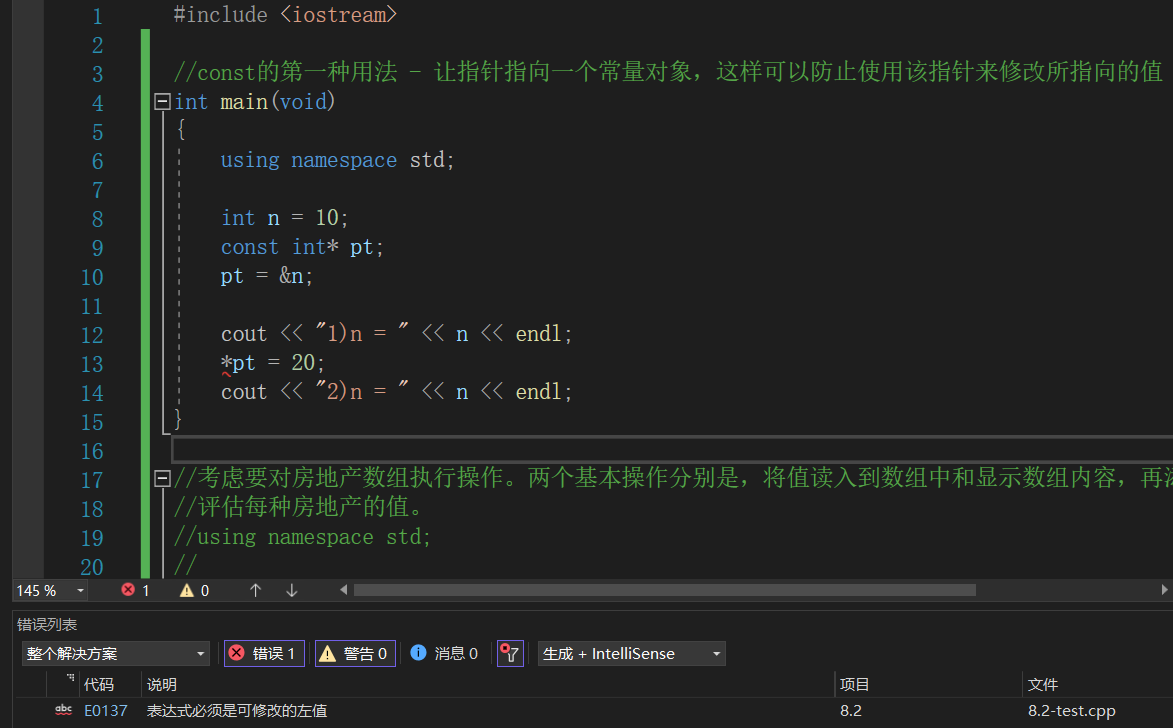
方法is\_open()是较新的内容，可以使用方法good()来代替。

**25.指针和const**

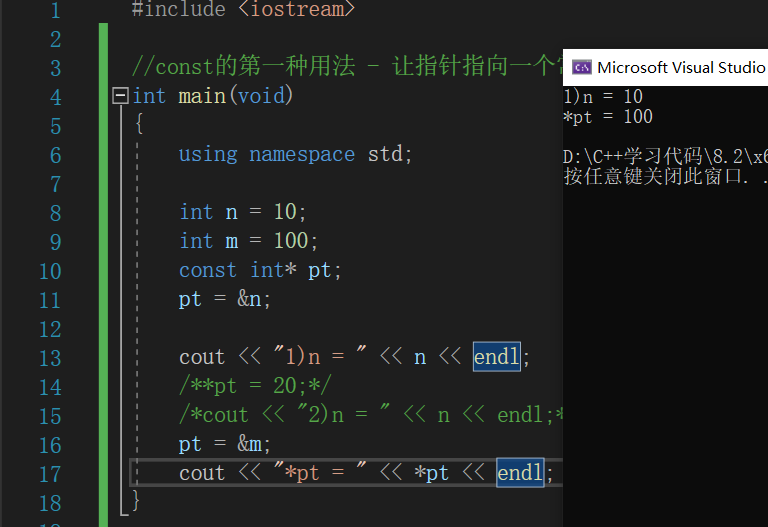
（1）const的第一种用法：让指针指向一个常量对象，这样可以防止使用该指针来修改所指向的值。

const int \*pt; 等价于 int const \*pt;

代码案例：



const修饰的是\*pt，因此不能通过\*pt去修改n的值。但却可以修改pt指向的内容，代码案例如下：



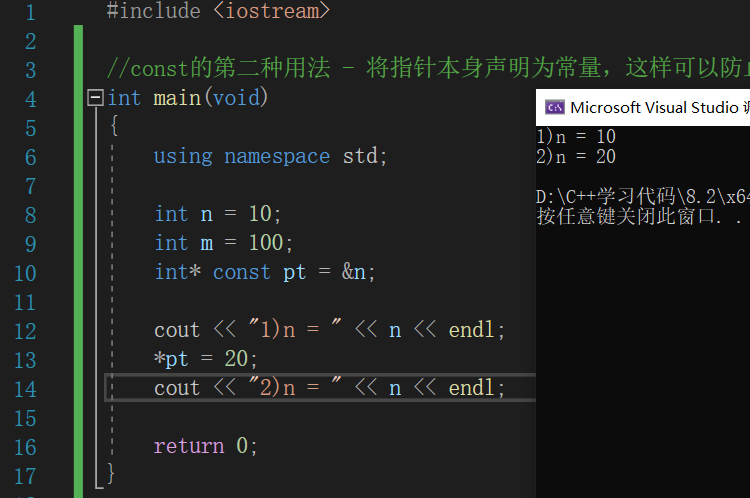
可以修改pt指针的指向，将它指向m的地址。

总的来说，第一种用法是const在\*前面,修饰的就是\*pt，不可以修改指针指向的值，但可以修改指针的位置。

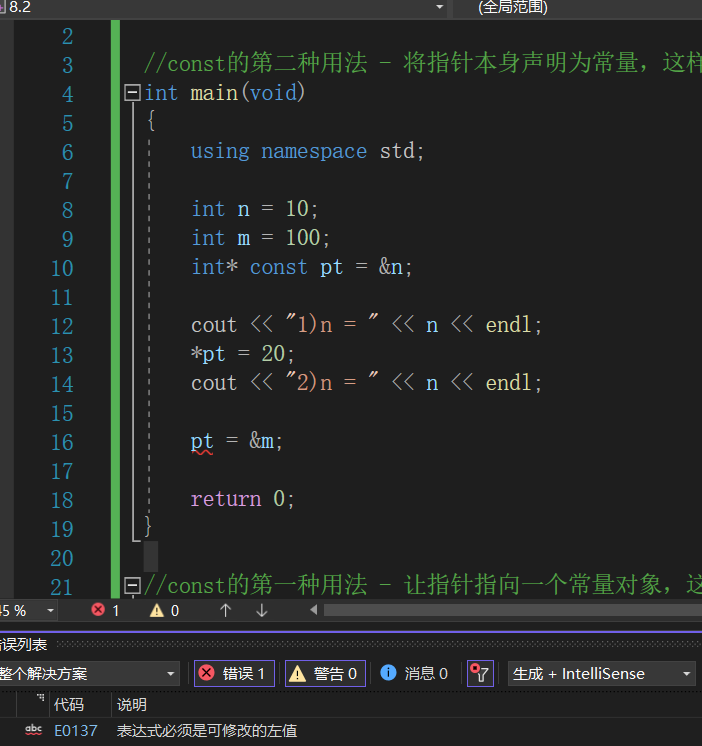
（2）const的第二种用法：将指针本身声明为常量，这样可以防止修改变指针的位置。

Int \*const pt;

代码案例：



因为const修饰的是pt，因此可以通过\*pt是去修改指针指向的值。但却不能修改指针的位置，代码案例如下：



可以看到第16行出错，就是因为const修饰了pt，因此不能对pt指针进行修改。

总的来说，第二种用法是const在\*的后面，修饰的是pt指针本身，因此可以修改指针指向的值，但不能修改指针的位置。

（3）使用const的原因

①这样可以避免由于无意间修改数据而导致的编程错误

1. 使用const使得函数能够处理const和非const实参，否则将只能接受非const数据。如果条件允许，则应将指针形参声明为指向const的指针。

**26.atan()函数**

double atan(double x);

double atan2(double y, double x);

传递的参数：x表示横坐标的值，y表示纵坐标的值

返回值：返回的是反正切值以弧度表示(第一个参数为y)

0-360表示角度：返回值 \* (180 / π)

**27.函数指针**

double (\*pf)(int); //pf指向一个参数为int类型，返回值为

//double类型的函数

process(think); //在process内部使用think()函数

thought(think()); //首先调用think函数，再将think的返回

//值传递给thought()。

pf扮演的角色与函数名相同，因此使用(\*pf)时，只需将它看做函数名即可。

例：

double pam(int);

double (\*pf)(int);

pf = pam; //让pf 指针指向pam函数

double x = pam(4); //使用函数名来调用pam()

double y = (\*pf)(5); //使用指针来调用pam()

C++也允许像使用函数名那样使用pf：

double y = pf(5);

(\*pa[2])(av, 3) \*pa[2](av, 3)

f3(av, 3) \*f3(av, 3)

代表第三个元素地址 代表第三个元素的值

28.内联函数

适用于代码执行时间短但调用时间长的程序

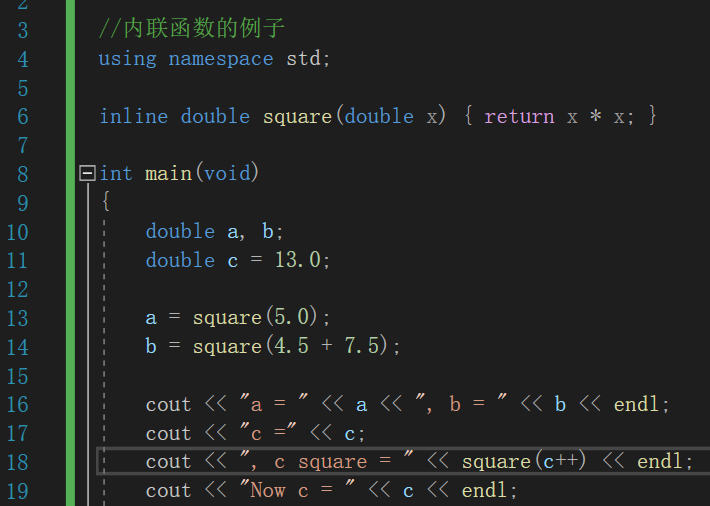
要使用内联函数，必须采取下述措施:

1. 在函数声明前加上关键字inline
2. 在函数定义前加上关键字inline

通常的做法是省略原型，将整个定义(即函数头和所有函数代码)放在本应提供原型的地方。

程序员将函数作为内联函数时，编译器并不一定会满足这种要求。它可能认为该函数过大或注意到函数调用了自己(内联函数不能递归)，因此不将其作为内联函数。

代码案例：



29.默认参数

将值赋给原型中的参数。例如：

char \*left(const char\* str, int n = 1);

默认参数值是初始化值，因此上面的原型将n初始化为1.如果省略参数n，则它的值将为1；否则，传递的值将覆盖1。

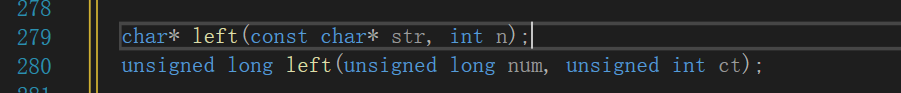
默认参数能够使用不同数目的参数调用同一个函数

默认参数必须从右到左设置，即如果有两个参数，想为第一个参数设置默认值的话，必须给第二个参数也设置默认值。

30.函数重载(多态)

函数重载能够使用多个同名的函数

函数重载的关键是函数的参数列表----也称为函数特征标。如果两个函数的参数数目和类型相同，同时参数排列顺序也相同，则称它们的特征标相同，而变量名是无关紧要的。C++允许定义名称相同的函数，条件是它们的特征标不同。如果参数数目和/或参数类型不同，则特征标也不同。例如：

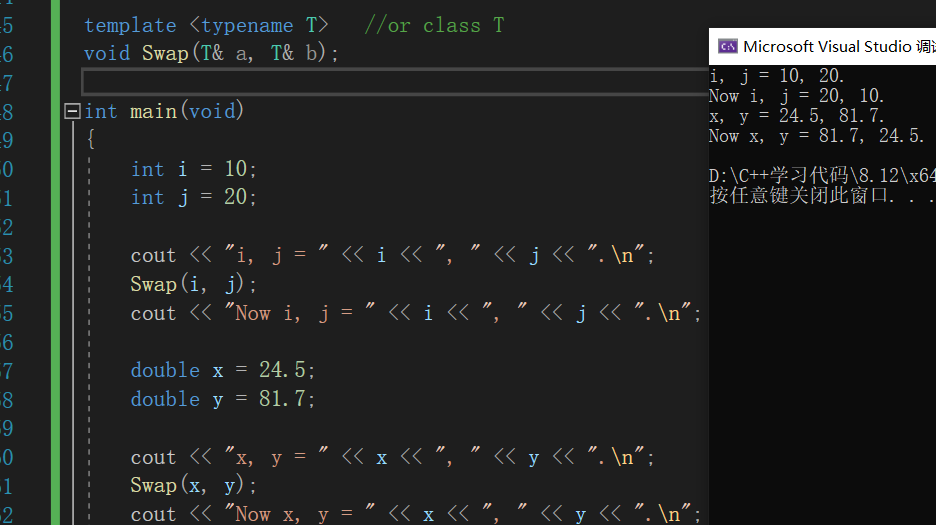


该函数名称相同，参数的类型不同。

31.函数模板

在函数定义前，需加上template <typename AnyType>（template<typename T>）

例如：



只用一个函数模板便将int类型参数和double类型的参数进行了交换，若不使用模板，需要分别编写以int类型为参数的函数和double类型为参数的函数。

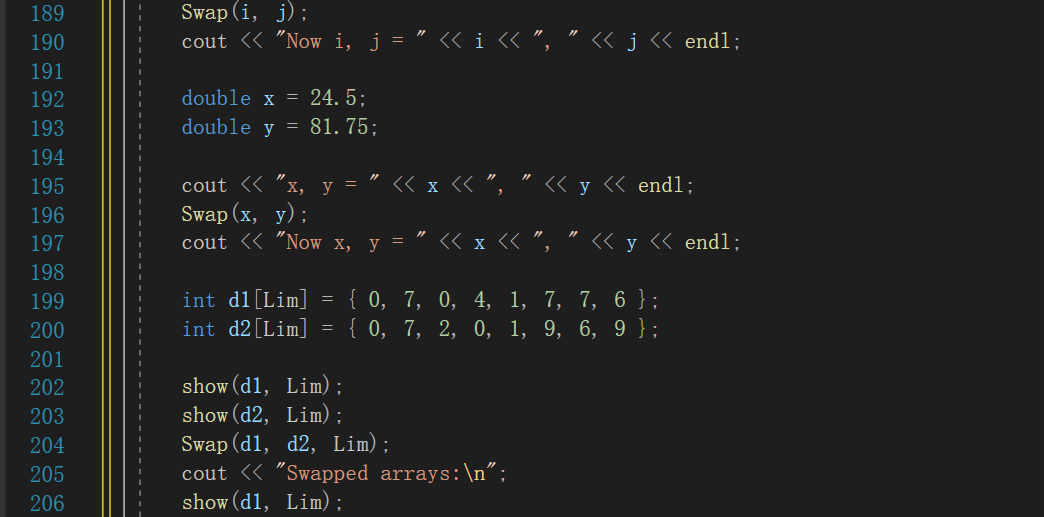
如果需要将同一种算法用于不同类型的函数，请使用模板。

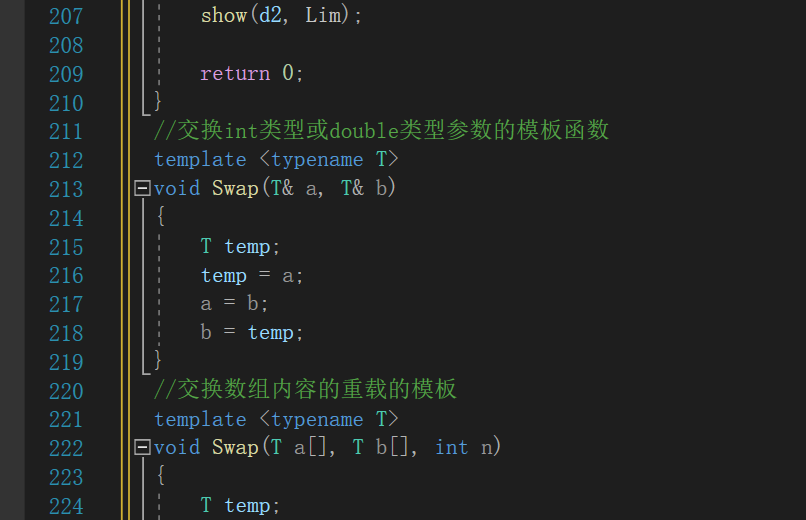
32.重载的模板

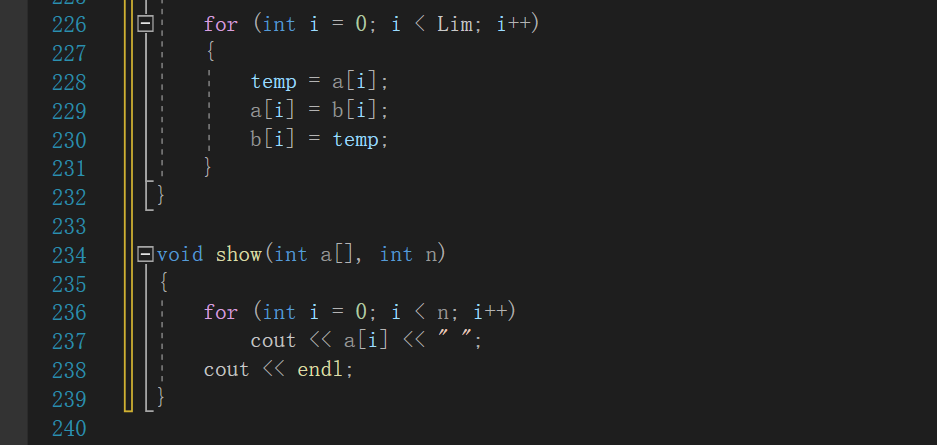
与常规重载一样，被重载的模板的函数特征标必须不同

代码案例：





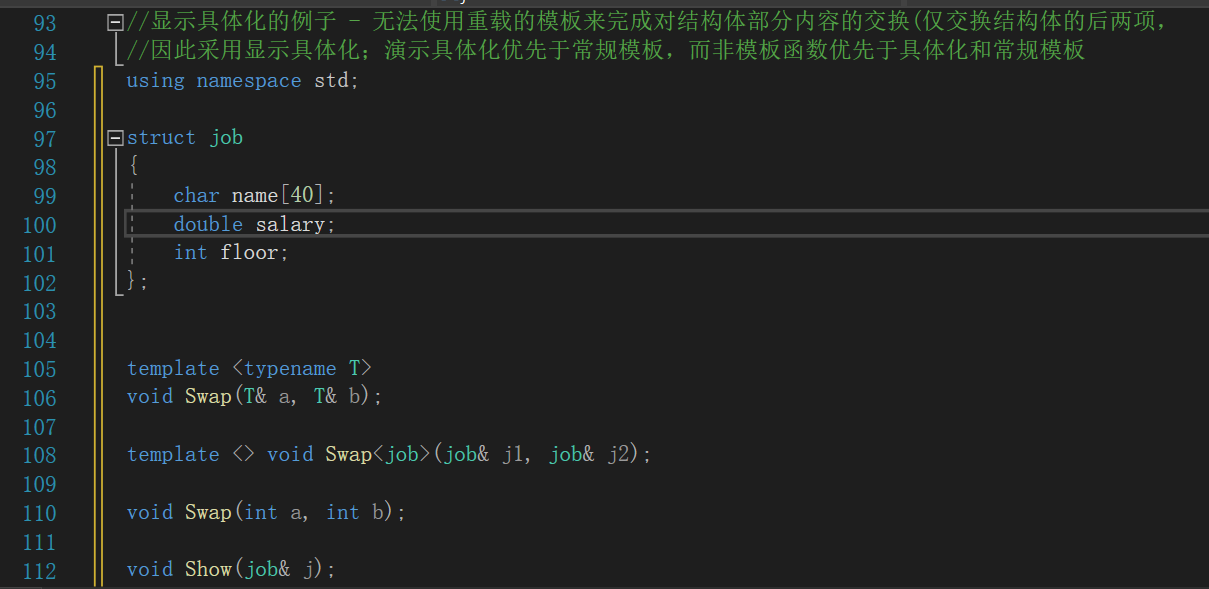




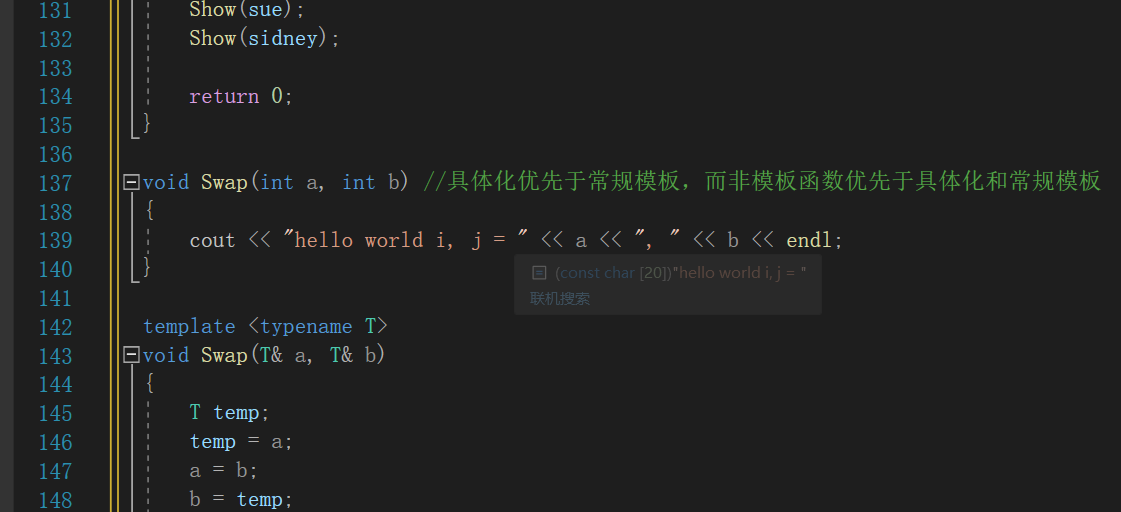
在函数声明部分，有两个同名的函数模板，第一种是以基本类型为参数的函数模板，第二个是以数组为参数的函数模板（也是第一种函数模板的重载）。

33.显示具体化

代码案例：



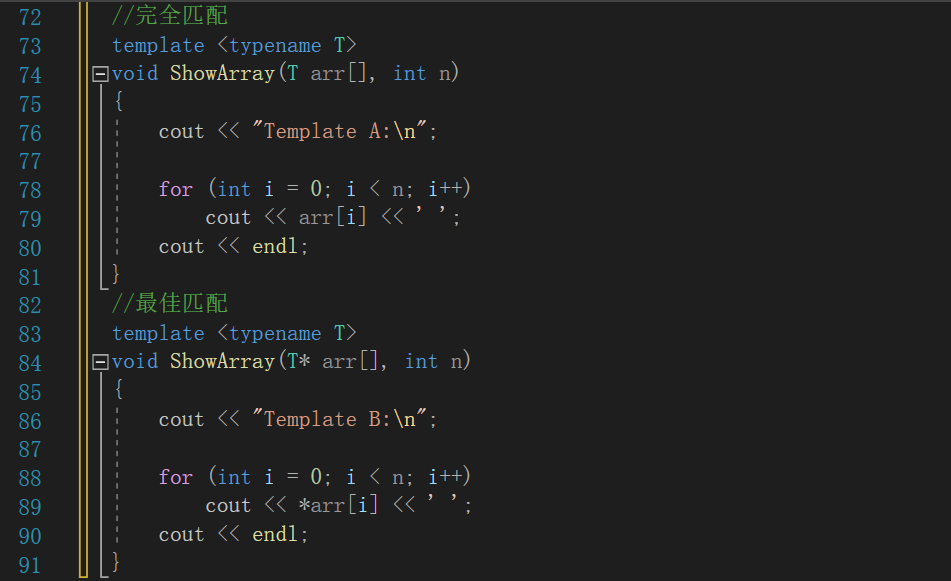
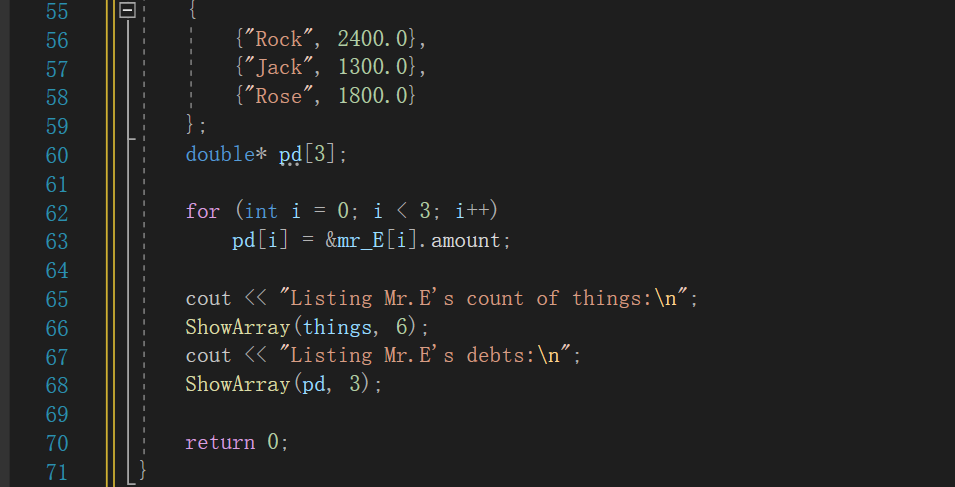
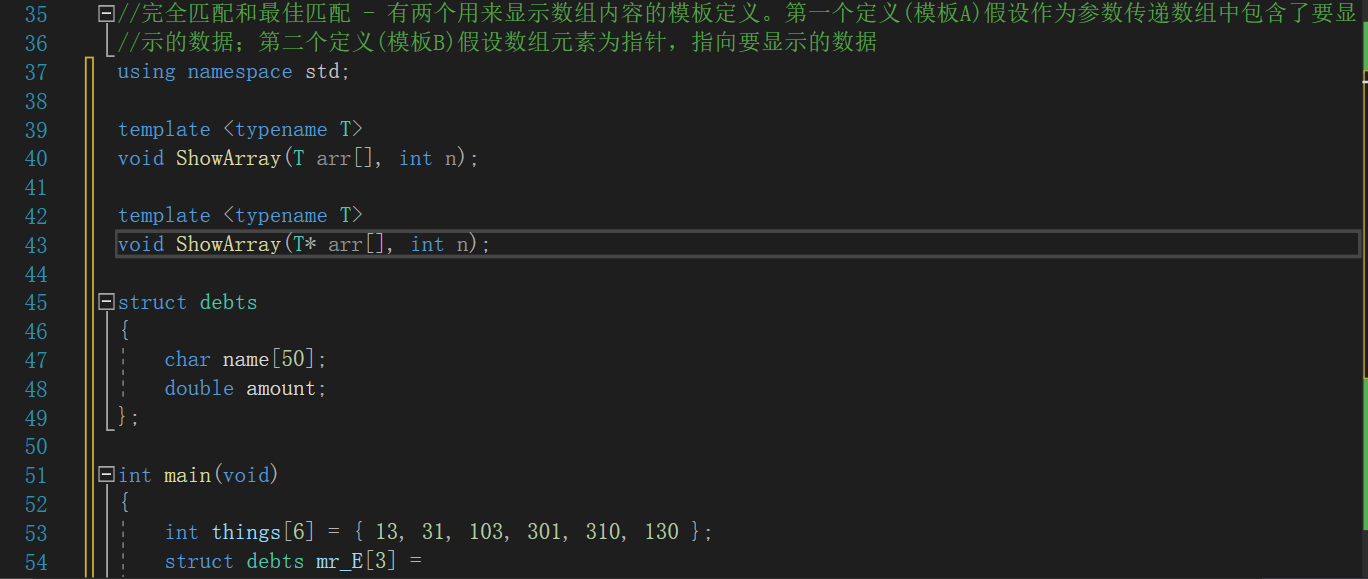






34.完全匹配和最佳匹配

代码案例：



在此程序中，对于pd来说，第一个和第二个模板都是其完全匹配，但第二个模板是其最佳匹配，因为其T只需转换为double类型即可，而第一个模板中T需要转换为double\*。

35.关键字decltype

例如：

//关键字decltype

//第一步，如果expression是一个没有括号括起的标识符，则var的类型与该标识符的类型相同:

// decltype(expression) var;

double x = 5.5;

double y = 7.9;

double& rx = x;

const double\* pd;

decltype(x) w; // w is type double

decltype(rx) u = y; //u is type double &

decltype(pd) v; //v is type const double\*

//第二步：如果expression是一个函数，则var的类型与函数的返回类型相同

long indeed(int);

decltype(indeed(3)) m; //m is type long

//第三步：如果expression是用括号括起来的标识符，则var为指向其类型的引用

double xx = 4.4;

decltype((xx)) r2 = xx; // r2 is double&

decltype(xx) w = xx; // w is double;

//第四步：如果前面的条件都不满足，则var的类型与erpression的类型相同

int j = 3;

int& k = j;

int& n = j;

decltype(j + 6) i1; // i1 type int

decltype(100L) i2; //i2 type long

decltype(k + n) i3; //i3 type int

//虽然k+n都是引用，但表达式k+n不是引用；它是两个int的和，因此类型为int

**36.另一种函数声明语法 - 后置返回类型**

template <typename T1, typename T2>

? type ? gt(T1 x, T2 y)

{

...

return x + y;

}

//在此例子中无法预知将x和y相加得到的类型。也无法将返回类型设置为decltype(x + y)，因为此时还未声明x和y参数，

//它们不在作用域内。为解决此问题，可以将函数声明定义成如下类型：

auto h(int x, float y) -> double;

//这将返回类型移到了参数声明后面。->double被称为后置返回类型。其中auto是一个占位符，表示后置返回类型提供的

//类型，通过与decltype结合使用，便可以给gt()指定返回类型，如下所示：

template <typename T1, typename T2>

auto gt(T1 x, T2 y) -> decltype(x + y)

{

...

return x + y;

}

**37.static限定符**

用static去修饰一个全局变量，那么该变量只能在该文件中使用，并会覆盖相应的全局定义。

38.构造函数

为何使用构造函数：C++的目标之一是让使用类对象就像使用标准类型一样，然而，普通的方法无法像初始化int或结构那样来初始化类的对象。为此，

C++提供了一个特殊的成员函数---类构造函数，专门用于构造新对像，自动进行初始化。

构造函数的原型和函数头有一个有趣的特征---虽然没有返回值，但没有被声明为void类型。实际上，构造函数没有声明类型。

非默认构造函数 – 用户自己定义的构造函数

代码例子：

构造函数的声明部分：

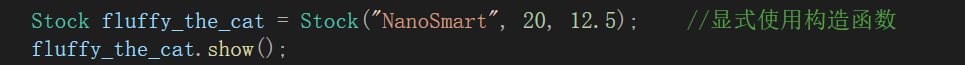


构造函数的定义部分：

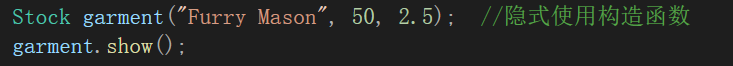


非默认构造函数的使用方法分为两种：

* 1. 显式使用的方式：



* 1. 隐式使用的方式：



默认构造函数 – 是在未提供显式初始值时，用来创建对象的构造函数。也就是说，它是用于下面这种声明的构造函数.

代码例子：



如果没有提供任何构造函数，则C++将自动提供默认构造函数。当且仅当没有定义没有定义任何构造函数时，编译器才会提供默认构造函数。

定义默认构造函数的方式有两种

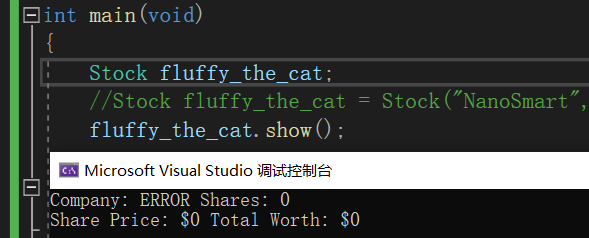
1. 给已有构造函数的所有参数提供默认值：

代码例子：

在头文件声明部分为形参赋值，无需更改定义部分的内容



运行结果如下：

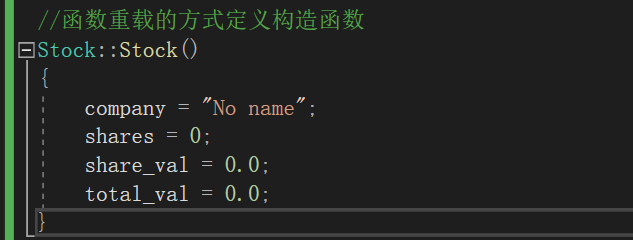


1. 通过函数重载来定义一个构造函数----一个没有参数的构造函数

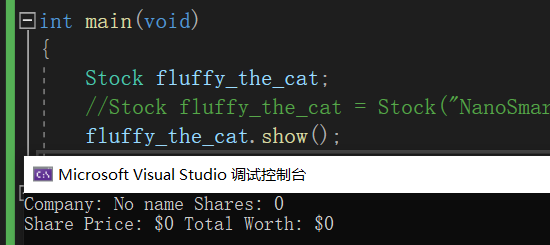
声明部分：



定义部分：



运行结果：



39.this指针

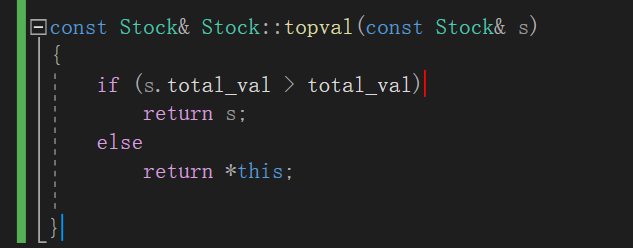
this指针指向用来调用成员函数的对象(this被作为隐藏参数传递给方法)。这样，函数调用stock1.topval(stock2)将this设置为stock1对象的地址，使得这个指针可用于topval()方法。同样，函数调用stock2.topval(stock1)将this设置为stock2对象的地址。一般来说，所有的类方法都将this指针设置为调用它的对象的地址。确实，topval()中的total\_val只不过是this->total\_val的简写，通过指针来访问成员，这也适用于类成员。

代码案例：

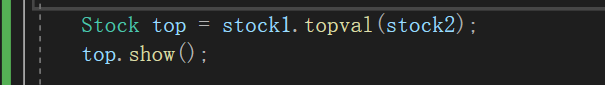
topval函数的声明



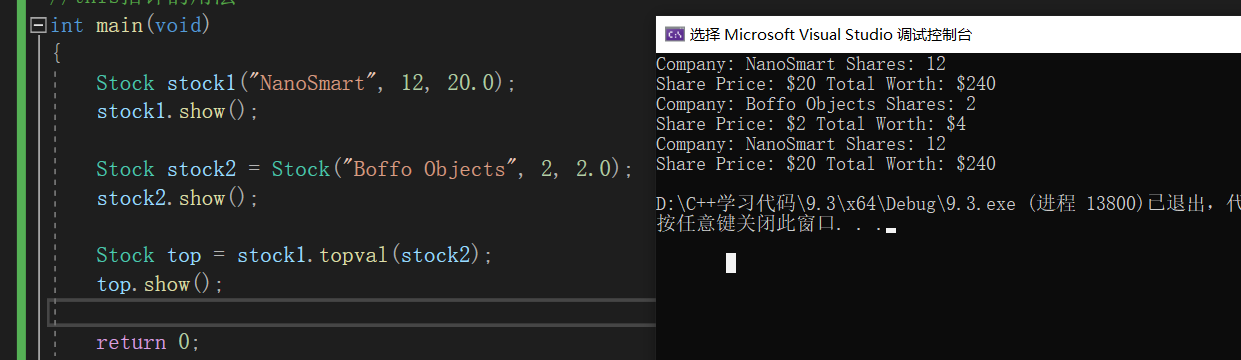
topval函数的定义以及this指针的用法



topval在主函数中的调用



其运行结果



40.运算符重载

operatorop(argument-list)

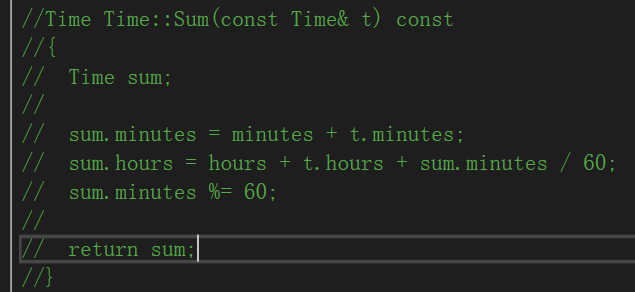
例如：oprerator+()重载+运算符，operator\*()重载\*运算符。op必须是有效的C++运算符。

首先，看一下没用运算符重载使两个类的相加的例子：

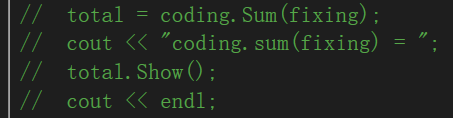
其函数在头文件中声明：



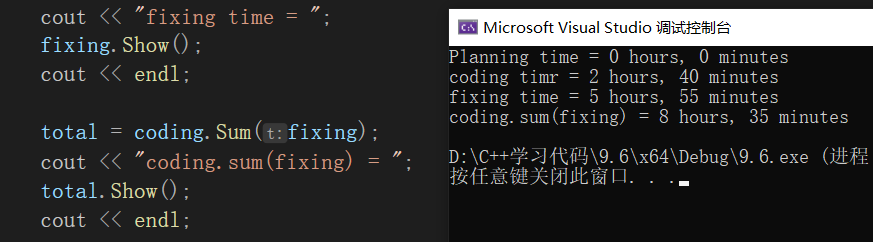
其定义部分如下：



调用部分：



运行结果为：



使用运算符重载：

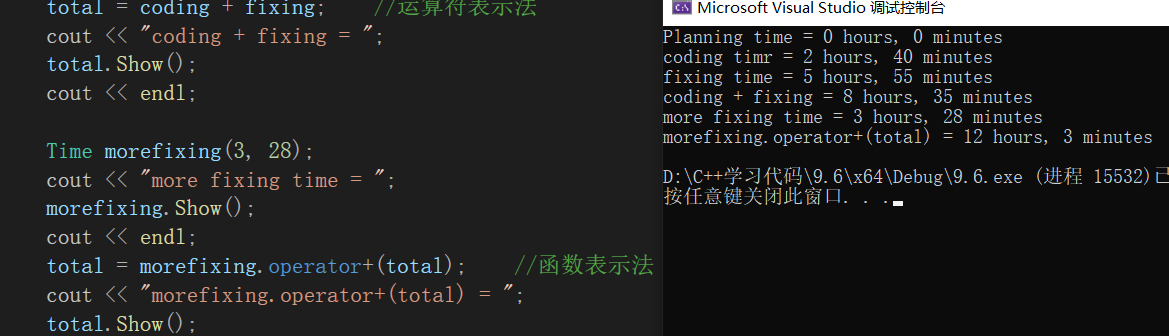
其声明部分：



其定义部分：



调用部分（两种表示法）和运行结果：

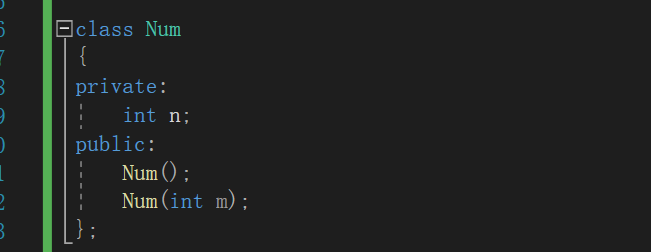


41.运算符重载限制

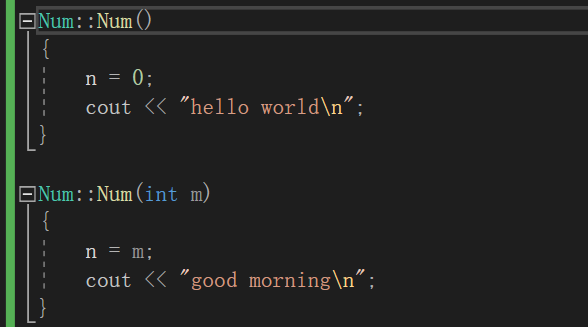
在程序中，若没有对=进行运算符重载，那么在主函数中若出现=运算符，此时程序将会在类中寻找合适参数的构造函数

例如：

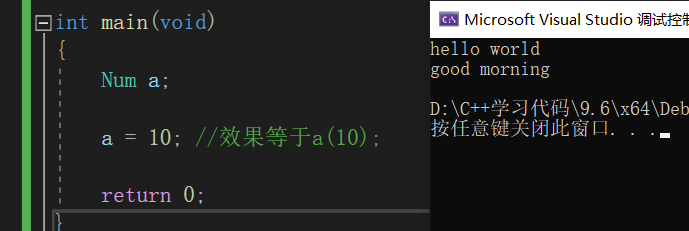
类的声明部分：



类的定义部分：



主函数以及运行结果：

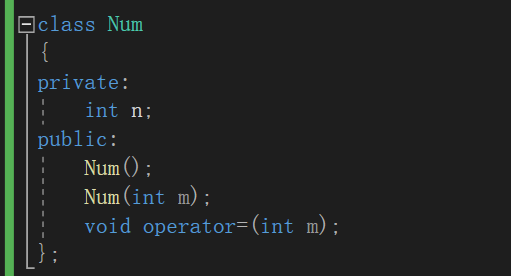


可以看出，按照正常逻辑来理解的话，a是一个类对象，而10则是一个int型的整数，如果将10赋给a,将出现类型不匹配的问题，但实际上程序确是正确运行了出来。其原因就是因为，=运算符在成员函数中有一个默认的重载，虽然用户没有自己定义，默认的=运算符会使这条语句去构造函数中寻找形参与右操作数相匹配的一个构造函数，可以看出与此条语句相匹配的构造函数是一个用户自定义的构造函数，因此打印出good morning。

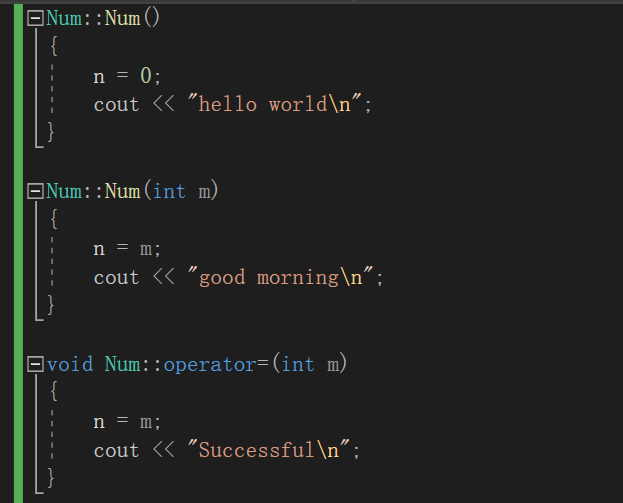
而类当中有对=运算符重载的情况时，在主函数中使用的=运算符则会调用类中自定义了的=运算符重载函数。

代码例子：

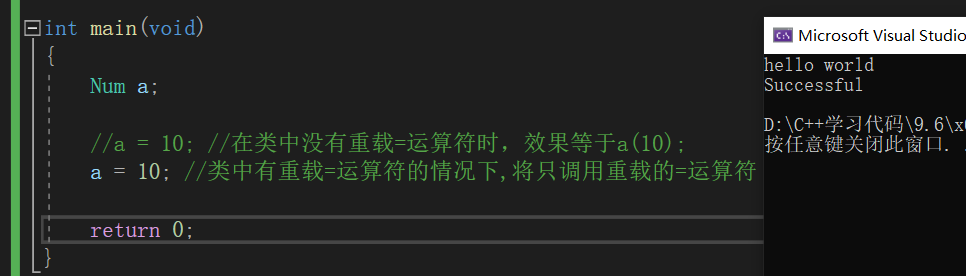
此时类的声明：



类的定义：



主函数以及运行结果：



通过上面的例子，可以得到运算符重载的限制：

大多数运算符都可以通过成员或非成员函数进行重载。

比如：+、-、\*、/运算符等。

但以下的运算符只能通过成员函数进行重载：

* =：赋值运算符
* ()：函数调用运算符
* []：下标运算符
* ->：通过指针访问类成员的运算符。

这些运算符之所以只能通过成员函数进行重载的原因是，在类中用户即使不自定义重载这些运算符，程序也会有默认的运算符重载，就如上面的=运算符一样，而若是在类之外（即非成员函数）使用这些运算符的重载，这个运算符重载将会是全局的，这将导致程序不知道选择类中的还是非成员定义的运算符重载，使得程序产生错误。

在C++规定中，不能重载以下的运算符：

* sizeof：sizeof运算符。
* .：成员运算符。
* .\*：成员指针运算符。
* ::：作用域运算符。
* ?:：条件运算符等。

41.构造函数的方便之处

使用场景：如果方法通过计算得到一个新的类对象，则应考虑是否可以使用类构造函数来完成这种工作，这样做不仅可以避免麻烦，而且可以确保新的对象是按照正确方式创建的。

