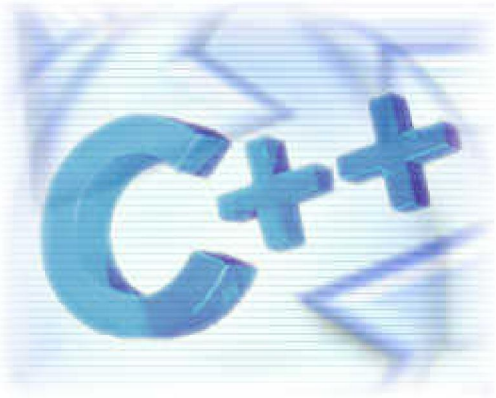
1. 一个月挑战 C++

作者：管宁 搜集整理：三尺寒冰

中国防黑网出品



2. 《挑战 30 天 C++入门极限》引言

作为一个长篇的 C++入门教程，无论如何也应该有这么个引言，可是文笔并不好的我，想了很久也不知道该如何写。。。。。。

仔细想想，与其把这篇短文当作教程的引言，其实它更应该是一篇引导初学者步入 C ＋＋殿堂的策论。

几年并不算很长的编程经验，使我感触颇深，痛苦迷茫,以及成功后的喜悦一直交替着伴随我，爱好编程的我，在学习期间也被很多很多学习的困难疑惑所阻挡，甚至不只一次的想放弃。

让我初次领略到作为一个程序员应该有快乐与喜悦的是 VISUAL BASIC 语言，这些年为了工作，也学习过很多其它的语言，包括时下正在流行的 JAVA 与 C#等等，在这里不得不说的是，作为一个程序员，一个能够适应当前中国工作环境的程序员，你要学的还远不止这些，包括数据系统，等等，等等。

很早就听人说过，如果你是一个程序员，如果你没有学过 C 语言，那么就不能算做是一个真正的程序员，这句话也许有点夸张，不过当你学习过它以后就不得不承认这句话真的有那么几分道

理，C++语言是由 C 语言发展而来的一种新的支持面向对象的语言，从一开始接触它，它的魅力就深深的吸引着我，起初我和很多想要学习它的人一样，觉得它很难懂，当时为了看懂，想明白一些现在看来真的很容易的例子时，真的有点让我抓狂。

作为一种灵活性高，体系庞大，支持面向对象思想的高级语言，C++的确比其它语言更难学习，很多正在学习它的在校大学生，以及很多正在从事编程工作想自学它的程序员，对于它的入门及各必要知识点掌握苦恼不已。

其实学习 C++和学习其它知识一样，并没有很多特别的要求，需要的只是那么一点点耐心，那么一点点忍耐力，以及遇到困难挫折不轻易屈服的精神,更重要的一点是你必须有一本好的，适合你的入门书籍指引你逐渐进步，面对书店里种类繁多，出名的不出名的 C ++教材，不光是初学者就连我也同样觉得眼花缭乱无从选择，很多优秀的国外 C ++教材有着过于系统的知识面以及跳跃式阅读的方法，这对初学者，尤其是从小就习惯从头到结尾阅读书籍的中国人来说是不合适的，国内的教材多以大学教材为主，这些书中的例子多以大学数学，或者工程数学举例，这样一来就把很多数学并不很好的读者拒之门外了。

在行内，很多人是不愿意和你分享学习经验的，一来工作任务繁重，再着很多人觉得这样会给自己带来更多的竞争对手。这些我都亲身经历过，正是因为此类的种种感受，于是我由内心而发，真心实意的愿意帮助那些正在学习 C ＋＋语言想入门的朋友们。

从一开始学习计算机语言，几乎所有的人都会问到，我该如何入门，入门后又该如何继续持久的进步下去呢？首先我要说的是，计算机语言的入门，无论是 C／C ＋＋还是 JAVA、 C#，他们都和高等数学没有太多联系，计算机语言是一种逻辑的文字描述，体现逻辑的是思想，当你通过理解一些计算机语言所描述的,并不算难的逻辑问题后，你就已经掌握了语言本身，并且真正入门了。

当然，计算机语言毕竟是和数学有密切联系的产物，在计算机世界里他们彼此依赖谁也离不开谁，当你正确理解编程思想，掌握必要知识点，入门之后，如果想做一个优秀的程序员而不是单单做一个程序的"拼装工"，你就不得不对数学知识进行进一步的系统学习。 此长篇 C++入门教程正是指引读者迈入 C++殿堂的初级读本，目的只是帮助读者快速入门，掌握必要的知识点，为了让教程通俗易懂，容易理解，适合自学，笔者为此下了不少工夫，同时也和我的一些朋友讨论过，意见中值得注意的是，一个作者如果没有教学，就只知道摸索一种适合自己的学习方法,对于语言学习的难点重点，关键是如何将复杂的问题用朴素的"俗"文化来写，

针对这些问题笔者笔者是认真反思过的，所以全书完全站在初学读者的思考的角度来写，教程中完全不包括任何难以理解的高等数学、工程数学问题，及C++标准库知识，着重对各入门的难点重点详细讲解分析，相信认真学习并读过它的人绝对不会觉得是浪费时间。

对于Ｃ＋＋学习，创建一条由入门到深入最后到精通的可持续学习并不段发展的道路概括起来可以是以下顺序。

１．学习一些基本的Ｃ语言知识，例如：什么是变量，什么是函数

２．Ｃ＋＋语言入门，基本知识点的掌握。

３．高等数学及工程数学的系统学习,结合实际了解并使用C++的各类常用的标准库。此外平时可以找一些广受好评的具备一定深度的C++教材看一下,进一步理解Ｃ++的高级编程精髓，以及看一些计算机原理和数据结构方面的书籍。

４．学习包括在各类操作系统下编程的必要知识，以Windows操作系统为例，需要学习掌握WIN API，以及高效开发的ＭＦＣ，ＶＣＬ等在内的其它知识。

千里之行始于足下，路虽然很长，困难也会很多，不过你一旦入门，一定会觉得面前豁然开朗，会不段的激励着你学习下去的。读者们如果在学习过程中有不懂的问题可以来我的站点提问,我会尽力解答的。站点地址：http://www.cndev-lab.com

写到这里笔者也呼吁更多的程序高手们，敲起你的键盘，把你的优秀的经验和思想奉献出来，帮助更多需要的人，毕竟思想是需要的是沟通的，知识是需要的是共享，快乐需要的是传递的。

最后在这里要感谢的是我的家人，我的父亲：管苗青、母亲：汪秋霞。多年来父亲母亲给我的帮助很大，一直耐心鼓励着我学习，母亲对我学习上的帮助已及她优秀的自学经验思想是我一生中最重要的财富。

作者：管宁

二零零五年三月十七日凌晨三点半

3. 在 c/c++中利用数组名作为函数参数传递排序和用指针进行排序的例子。

以下两个例子要非常注意，函数传递的不是数组中数组元素的真实值而是数组在内存中的实际地址。

```
#include <stdio.h>

void main(void)
{

void reversal();
static int a[10] = {0,1,2,3,4,5,6,7,8,9}; /* 建立一个数组并初始化 */
int i;
for (i=0;i<10;i++)
{
printf("%d ",a);
}
printf("\n");
reversal(a,10); /* 调用自定义涵数进行反向显示排序,并把数组 a 的起始地址传送给形式参数 x */

for (i=0;i<10;i++)
{
printf("%d ",a);
}
printf("\n");

}

void reversal(x,n)
int x[],n; /* 定义形式参数 */
{
int m=(n-1)/2; /* 计算 10 个数需要循环几次,因为是两两调换第一个数组是 x[0]故应该是 int(9/2) */
int temp,i,j; /* 建立零时变量 temp 用于每次交换处理时零时存储 x 的值 */
for (i=0;i<=m;i++)
{
j=n-1-i; /* 反向计算出被调换的数组下标,例如 x[0] 对应的 x[n-1-i]就是 x[9] */
temp=x;
x=x[j];
x[j]=temp;
```

```
}
}
```

/* 次题需要注意的是:这里由于 a[10]和 x[10]是共同享内存地址位的所以进行交换后 a[10]的实际值也就发生了改变 */

```
#include <stdio.h>

void main(void)
{

void reversal();
static int a[10] = {0,1,2,3,4,5,6,7,8,9}; /* 建立一个数组并初始化 */
int i;
for (i=0;i<10;i++)
{
printf("%d ",a);
}
printf("\n");
reversal(a,10); /* 调用自定义涵数进行反向显示排序,并把数组 a 的起始地址传送给形式参数 x */

for (i=0;i<10;i++)
{
printf("%d ",a);
}
printf("\n");

}

void reversal(x,n)
int *x,n; /* 定义 x 为指针变量 */
{
int temp,*p,*i,*j; /* 这里需要注意的是 temp 用与交换的时候临时存储数据的 */
i = x; /* 利用指针变量 i 存储数组 a 的起始地址 */
p = x + ((n-1)/2); /* 计算最后一次循环的时候数组 a 的地址 */
j = x + n - 1; /* 计算数组 a 也就是 a[9]的结束地址好用于交换 */
for (;i<=p;i++,j--) /* 利用循环和指针进行数组元素值的交换 */
{
temp=*i; /* 用 temp 临时存储*i 也就是循环中 a 实际的值 */
*i=*j;
*j=temp;
}
```

```
}

/* 此例同样要注意到利用指针进行数组的操作同样改变了实际数组各元素的值 */
```

4. c/c++中指针学习的两个绝好例子

对于众多人提出的 c/c++中指针难学的问题做个总结：

指针学习不好关键是概念不清造成的，说的简单点就是书没有认真看，指针的学习犹如人在学习饶口令不多看多学多练是不行的，下面是两个很经典的例子，很多书上都有,对于学习的重点在于理解*x 和 x 的理解，他们并不相同，*x 所表示的其实就是变量 a 本身，x 表示的是变量 a 在内存中的地址，如果想明白可以输出观察 cout<<*x"|"x;，当定义了 int *x;后对 x=&a 的理解的问题。仔细阅读和联系下面的两个例子我想指针问题就不是难点了！

```
#include <stdio.h>

main()
{
int a,b; /* 定义 a,b 两个整形变量用于输入两个整数 */
int *point_1,*point_2,*temp_point; /* 定义三个指针变量 */
scanf("%d,%d",&a,&b); /* 格式化输入 a,b 的值 */
point_1=&a; /* 把指针变量 point_1 的值指向变量 a 的地址 */
point_2=&b; /* 把指针变量 point_2 的值指向变量 b 的地址 */
if (a<b)
{
   temp_point=point_1; /* 这里的 temp_point 是用于临时存储 point_1 的值也就是变量 a 的地址的 */
   point_1=point_2; /* 把 point_2 的值赋予 point_1 */
   point_2=temp_point;
   /* 由于 point_1 的值已经改变无法找到,利用前面临时存储的也就是 temp_point 找回原 point_1 的值赋予 point_2,打到把 point_1 和
point_2 值对换的目的*/
}
printf("%d,%d",*point_1,*point_2); /* 利用*point_1 和*point_2 也就是分辨指向 b 和 a 的方法把值显示自爱屏幕上 */
}

/* 此题需要注意和了解是的此法并没有改变变量 a,b 的值只是利用指针变量分别存储 a 和 b 的地址,然后再把那两个指针变量的值对换一
下其实就是存储在
指针变量里面 a 与 b 的地址对换,在利用*point_1 和*point_2 的方式把调换后的值显示出来这里的*point_1 实际就是 a,此中算法并非真的
改变 a,b 的值,而是
利用指针进行地址交换达到大小排序的目的.
*/
```

```
#include <stdio.h>

main()
{
int a,b; /* 定义 a,b 两个整形变量用于输入两个整数 */
int *point_1,*point_2; /* 定义三个指针变量 */
scanf("%d,%d",&a,&b); /* 格式化输入 a,b 的值 */
point_1 = &a; /* 把指针变量 point_1 的值指向变量 a 的地址 */
point_2 = &b; /* 把指针变量 point_2 的值指向变量 b 的地址 */
compositor(point_1,point_2); /* 调用自定义的排序涵数,把 a,b 的地址传递给 point_1 和 point_2 */
printf("%d,%d",a,b); /* 打印出 a,b 的值 */
}

static compositor(p1,p2)
int *p1,*p2; /* 定义形式参数 p1,p2 为指针变量 */
{
int temp; /* 建立临时存储变量 */
   if (*p1<*p2) /* 如果*p1<p2,注意这里的*p1 和*p2 其实就是 a 和 b */
   {
      temp = *p1; /* 利用变量 temp 用于临时存储*p1 和就是 a 的值 */
      *p1 = *p2; /* 将*p1 的值也就是 a 的值换成*p2 的值也就是 b 的值,等价于 a=b */
      *p2 = temp; /* 将*p2 的值也就是 temp 的值等价于 b=temp */
   }
}

/* 注意:此题与上题不同的是,直接改变了 a 于 b 的值达到真实改变的目的 */
```

5. 入门教程:C++中的 const 限定修饰符

const 修饰符可以把对象转变成常数对象，什么意思呢？

意思就是说利用 const 进行修饰的变量的值在程序的任意位置将不能再被修改，就如同常数一样使用！

使用方法是：

```
const int a=1;//这里定义了一个 int 类型的 const 常数变量 a;
```

但就于指针来说 const 仍然是起作用的，以下有两点要十分注意，因为下面的两个问题很容易混淆！

我们来看一个如下的例子：

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

void main(void)
{
const int a=10;
int b=20;

const int *pi;
pi=&a;
cout <<*pi << "|" << a <<endl;
pi=&b;
cout <<*pi << "|" <<b <<endl;
cin.get();
}
```

上面的代码中最重要的一句是 const int *pi
这句从右向座读作：pi 是一个指向 int 类型的,被定义成 const 的对象的指针;
这样的一种声明方式的作用是**可以修改** pi 这个指针**所指向的内存地址**却**不能修改指向对象的值**;

如果你在代码后加上*pi=10;这样的赋值操作是不被允许编译的！

好，看了上面的两个例子你对 const 有了一个基本的认识了，那么我们接下来看一个很容易混淆的用法！

请看如下的代码：

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

void main(void)
{
int a=10;

const int *const pi=&a;

cout <<*pi << "|" <<a <<endl;
cin.get();
}
```

上面的代码中最重要的一句是 const int *const pi
这句从右向座读作：pi 是一个指向 int 类型对象的 const 指针;

这样的一种声明方式的作用是你**既不可以修改 pi 所指向对象的内存地址也不能利用指针的解引用方式修改对象的值，也就是用 *pi=10 这样的方式**;

所以你如果在最后加上*pi=20，想试图通过这样的方式修改对象 a 的值是不被允许编译的!

结合上面的两点所说，把代码修改成如下形式后就可以必然在程序的任意的地方修改对象 a 的值或者是指针 pi 的地址了，**下面的这种写法常被用语涵数的形式参数,这样可以保证对象不会在涵数内被改变值!**

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

void main(void)
{
const int a=10;//这句和上面不同,请注意!

const int *const pi=&a;

cout <<*pi << "|" <<a <<endl;
cin.get();
}
```

6. 新手入门：C++下的引用类型

引用类型也称别名，它是个很有趣的东西。在 c++ 下你可以把它看作是另外的一种指针，通过引用类型我们同样也可以间接的操作对象，引用类型主要是用在函数的形式参数上，通常我们使用它是把类对象传递给一个函数。 引用对象采用类型名加上&符号和名称的方式进行定义。例如：(int &test;)，这里我们就定义了一个 int 类型的名为 test 的引用，但是 int &test;这样的方式是不能够被编译成功的，因为引用的定义必须同时给应用进行赋值操作，这里的赋值并不是说把变量的值传递给引用，而是把引用指向变量，写成这样就对了：(int &test=变量名;)。

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

void main(void)
{

int a=10;
int &test=a;
test=test+2;

cout << &a << "|" << &test << "|" << a << "|" <<test << endl;

cin.get();
}
```

观察并编译运行上面的代码你会发现&a 和&test 的地址显示是相同的，a 和 test 的值显示也是一样的！

结合前一个教程的内容我们来说一下 const 引用的相关内容，这里要特别注意，和前一个教程一样带 const 修饰的引用同样也容易混淆概念！

const 修饰如果用在引用上会有一个特别之处，它的奥妙就在于可以进行不同类型的对象的初始化，而这一切在普通变量操作上是不可能的下面我们来看一个例子：

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

void main(void)
{

int a=10;
//double &test = a + 1.2f; //这句就是错误的!
const double &test = a + 1.2f;
```

```
cout << &a << "|" << &test << "|" << a << "|" <<test << endl;

cin.get();
}
```

上面的代码足够说明问题了，这就是 const 修饰带来的好处，但是聪明的人会在输出的时候发现一个问题，就是 a 和 test 的值的输出不同，按照最先说的道理应该可以改变 a 的值呀，为什么在这里却有没有能够改变呢？

道理是这样的，const 修饰过后的引用在编译器内部是这样进行变化的。

```
int a=10;
const double &test = a + 1.2f;
```

这样的一段代码在编译器认为却是下面的方式进行的

```
int a=10;
int temp = a;
const double &test = temp + 12.f
```

这里其实是把 a 的值赋给了一个**临时** temp 变量，而后 test 获得的却是 temp+12.f 改变的是 temp 而不是 a，所以就出现了 a 和 test 显示的值不同的情况，这里要特别注意,这是一个很容易混淆的地方，在编写程序的时候要特别仔细，以免出现了问题却检查不出为什么！

7. 新手入门：C++中布尔类型

布尔类型对象可以被赋予文字值 true 或者 false，所对应的关系就是真与假的概念。

我们通常使用的方法是利用他来判断条件的真与假，例如下面的代码：

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

void main(void)
{
   bool found = true;
   if (found)
   {
      cout << "found 条件为真!" << endl;
   }

}
```

但是一些概念不清的人却不知道布尔类型的对象也可以被看做是一种整数类型的对象，但是他不能被声明成 signed，unsigned，short long，如果你生成(short bool found=false;)，那么将会导致编译错误。

其为整数类型的概念是这样的：

当表达式需要一个算术值的时候，布尔类型对象将被隐式的转换成 int 类型也就是整形对象， false 就是 0，true 就是 1，请看下面的代码！

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

void main(void)
{
bool found = true;
int a = 1;
cout << a + found << endl;
cin.get();
}
```

a+found 这样的表达式样是成立的，输出后的值为 2 进行了加法运算！

那么说到这里很多人会问指针也可以吗？回答是肯定的这样一个概念对于指针同样也是有效的，下面我们来看一个将整形指针对象当作布尔对象进行使用的例子：

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

void main(void)
{
   int a = 1;
   int *pi;
   pi=&a;

   if (*pi)
   {
      cout << "*pi 为真" << endl;
   }
   cin.get();
}
```

上面代码中的*pi 进行了隐式样的布尔类型转换表示为了真也就是 true。

8. 新手入门：C/C++中枚举类型(enum)

如果一个变量你需要几种可能存在的值，那么就可以被定义成为枚举类型。之所以叫枚举就是说将变量或者叫对象可能存在的情况也可以说是可能的值一一例举出来。

举个例子来说明一吧，为了让大家更明白一点，比如一个铅笔盒中有一支笔，但在没有打开之前你并不知道它是什么笔，可能是铅笔也可能是钢笔，这里有两种可能，那么你就可以定义一个枚举类型来表示它！

```
enum box{pencil,pen};//这里你就定义了一个枚举类型的变量叫 box，这个枚举变量内含有两个元素也称枚举元素在这里是 pencil 和 pen，分别表示铅笔和钢笔。
```

这里要说一下，如果你想定义两个具有同样特性枚举类型的变量那么你可以用如下的两种方式进行定义！

```
enum box{pencil,pen};

enum box box2;//或者简写成 box box2;
```

再有一种就是在声明的时候同时定义。

```
enum {pencil,pen}box,box2; //在声明的同时进行定义！
```

枚举变量中的枚举元素系统是按照常量来处理的，故叫**枚举常量**，他们是不能进行普通的算术赋值的，(pencil=1;)这样的写发是错误的，但是你可以在声明的时候进行赋值操作！

```
enum box{pencil=1,pen=2};
```

但是这里要特别注意的一点是，如果你不进行元素赋值操作那么元素将会被系统自动从 0 开始自动递增的进行赋值操作，说到自动赋值，如果你只定义了第一个那么系统将对下一个元素进行前一个元素的值加 1 操作，例如

```
enum box{pencil=3,pen};//这里 pen 就是 4 系统将自动进行 pen=4 的定义赋值操作！
```

前面说了那么多，下面给出一个完整的例子大家可以通过以下的代码的学习进行更完整的学习！

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

void main(void)
{
```

```
    enum egg {a,b,c};
    enum egg test; //在这里你可以简写成 egg test;

    test = c; //对枚举变量 test 进行赋予元素操作，这里之所以叫赋元素操作不叫赋值操作就是为了让大家明白枚举变量是不能直接赋予算数值的，例如(test=1;)这样的操作都是不被编译器所接受的，正确的方式是先进行强制类型转换例如(test = (enum egg) 0;)！

    if (test==c)
    {
        cout <<"枚举变量判断:test 枚举对应的枚举元素是 c" << endl;
    }

    if (test==2)
    {
        cout <<"枚举变量判断:test 枚举元素的值是 2" << endl;
    }

    cout << a << "|" << b << "|" << test <<endl;

    test = (enum egg) 0; //强制类型转换
    cout << "枚举变量 test 值改变为:" << test <<endl;
    cin.get();
}
```

看到这里要最后说一个问题，就是枚举变量中的枚举元素(或者叫枚举常量)在特殊情况下是会被自动提升为算术类型的！

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

void main(void)
{
    enum test {a,b};
    int c=1+b; //自动提升为算术类型
    cout << c <<endl;
    cin.get();
}
```

9. 新手入门：C/C++中数组和指针类型的关系

对于数组和多维数组的内容这里就不再讨论了，前面的教程有过说明，这里主要讲述的数组和指针类型的关系，通过对他们之间关系的了解可以更加深入的掌握数组和指针特性的知识！

一个整数类型数组如下进行定义：

```
int a[]={1,2,3,4};
```

如果简单写成：

```
a;//数组的标识符名称
```

这将代表的是数组第一个元素的内存地址，a;就相当于&a[0]，它的类型是数组元素类型的指针，在这个例子中它的类型就是 int*

如果我们想访问第二个元素的地址我们可以写成如下的两种方式：

```
&a[1];

a+1//注意这里的表示就是将 a 数组的起始地址向后进一位，移动到第二个元素的地址上也就是 a[0]到 a[1]的过程！
```

数组名称和指针的关系其实很简单，其实数组名称代表的是数组的第一个元素的内存地址，这和指针的道理是相似的！

下面我们来看一个完整的例子，利用指针来实现对数组元素的循环遍历访问！

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

void main(void)
{
   int a[2]={1,2};

   int *pb=a; //定义指针*pb 的地址为数组 a 的开始地址

   int *pe=a+2; //定义指针*pb 的地址为数组 a 的结束地址

   cout << a << "|" << a[0] << "|" << *(a+1) << "|" << pb << "|" << *pb <<endl;

   while (pb!=pe) //利用地址进行逻辑判断是否到达数组的结束地址
```

```
    {
        cout << *pb << endl;
        pb++; //利用递增操作在循环中将 pb 的内存地址不断向后递增
    }
    cin.get();
}
```

10.新手入门：关于 C++中的内联函数(inline)

在 c++中，为了解决一些频繁调用的小函数大量消耗栈空间或者是叫栈内存的问题，特别的引入了 inline 修饰符，表示为内联函数。

可能说到这里，很多人还不明白什么是栈空间，其实栈空间就是指放置程序的局部数据也就是函数内数据的内存空间，在系统下，栈空间是有限的，如果频繁大量的使用就会造成因栈空间不足所造成的程序出错的问题，函数的死循环递归调用的最终结果就是导致栈内存空间枯竭。

下面我们来看一个例子：

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

inline string dbtest(int a); //函数原形声明为 inline 即:内联函数


void main()
{
   for (int i=1;i<=10;i++)
   {
      cout << i << ":" << dbtest(i) << endl;
   }
   cin.get();

}

string dbtest(int a)//这里不用再次 inline,当然加上 inline 也是不会出错的
{
   return (a%2>0)?"奇":"偶";

}
```

上面的例子就是标准的内联函数的用法，使用 inline 修饰带来的好处我们表面看不出来，其实在内部的工作就是在每个 for 循环的内部所有调用 dbtest(i)的地方都换成了(i%2>0)?"奇":"偶"这样就避免了频繁调用函数对栈内存重复开辟所带来的消耗。

说到这里很多人可能会问，既然 inline 这么好，还不如把所谓的函数都声明成 inline，嗯，这个问题是要注意的，inline 的使用是有所限制的，inline 只适合函数体内代码简单的函数使用，不能包含复杂的结构控制语句例如 while switch，并且不能内联函数本身不能是直接递归函数(自己内部还调用自己的函数)。

说到这里我们不得不说一下在 c 语言中广泛被使用的#define 语句，是的 define 的确也可以做到 inline 的这些工作，但是 define 是会产生副作用的，尤其是不同类型参数所导致的错误，由此可见 inline 有更强的约束性和能够让编译器检查出更多错误的特性，在 c++中是不推荐使用 define 的。

关于内联函数的更多例子我就不一一举出了，灵活的使用也多靠学习者本身，我只在此抛砖引玉，让大家尽可能多的学习到 c++中的一些新的先进的特性知识点。

11.新手入门：C++中的函数重载

函数重载是用来描述同名函数具有相同或者相似功能,但数据类型或者是参数不同的函数管理操作的称呼。

我们来举一个实际应用中的例子来说明问题:我们要进行两种不同数据类型的和操作为了实现它，在 c 语言中我们就要写两个不同名称的涵数来进行区分例如:int testa(int a,int b)和 float testb(float a,floatb)，这样字是没有问题,但是总有一点不好，这么两个具备极其相似操作函数我们却起两个不同的名字，这样子不是很好管理，所以 c++为了方便程序员编写程序特别引入了函数重载的概念来解决此问题，我们看看如下的代码：

```
#include <iostream>
using namespace std;
int test(int a,int b);
float test(float a,float b);
void main()
{
cout << test(1,2) << endl << test(2.1f,3.14f) << endl;
cin.get();
}

int test(int a,int b)
{
return a+b;
}

float test(float a,float b)
{
return a+b;
}
```

在上面的程序中我们同样使用了两个名为 test 的函数来描述 int 类型和操作的和 float 类型和操作，这样一来就方便了程序员对相同或者相似功能函数的管理。

看了上面的解释很多人会问，这么一来计算机该如何来判断同名称函数呢？操作的时候会不会造成选择错误呢？

回答是否定的。c++内部利用一种叫做名称粉碎的机智来内部重命名同名函数,上面的例子在计算重命名后可能会是 testii 和 testff 他们是通过参数的类型或个数来内部重命名的，关于这个作为程序员不需要去了解它，说一下只是为了解释大家心中的疑问而已。好了，关于函数学重载的基础知识就说到这里，至于如何利用这个功能,就靠大家在日常的学习或者是工作中逐渐摸索了。

12.新手入门：C++中堆内存(heap)的概念和操作方法

堆内存是什么呢？

我们知道在 c/c++中定义的数组大小必需要事先定义好，他们通常是分配在静态内存空间或者是在栈内存空间内的，但是在实际工作中，我们有时候却需要动态的为数组分配大小，在这里 c 库中的 malloc.h 头文件中的 malloc()函数就为您解决了问题（bc 或者是在老的标准中是 alloc.h），它的函数原形是 void* malloc(size_t size)，在动态开辟的内存中，在使用完后我们要使用 free()函数来释放动态开辟的内存空间。

下面我们来看一个完整的例子：

```
#include <iostream>
#include <malloc.h>

using namespace std;
main()
{
   int arraysize; //元素个数
   int *array; //用于动态开辟数组的指针变量

   cin>>arraysize;
   array=(int*)malloc(arraysize * sizeof(int));//利用 malloc 在堆内存中开辟内存空间,它的大小是元素的个数乘以该数据类型的长度

   for(int i=0;i<arraysize;i++)
   {
      array[i]=i;
   }

   for(int i=0;i<arraysize;i++)
   {
      cout<<array[i]<<",";
   }
   cout<<endl;
   free(array);//利用 free 释放动态开辟的堆内存空间
   cin.get();
   cin.get();
}
```

这里要特别注意个地方就是：

```
array=(int*)malloc(arraysize * sizeof(int));
```

malloc()的函数原形本身是 void* malloc(size_t size)，由于动态分配的空间计算机并不知道是用来做什么的所以是无类型的，但你要把它用在动态的整形数组上的时候就要显式的转换成 int*了。

下面我们再介绍 c++所独有的开辟和释放堆内存空间的方法，new 修饰符和 delete 修饰符。

new 和 delete 修饰符的操作并不需要头文件的支持，这是 c++所独有的,new 操作要比 malloc 更为简单，直接说明开辟的类型的数目就可以了，delete 使用的时候如果是数组那么必须使用 delete[]。

```
#include <iostream>

using namespace std;
main()
{
   int arraysize; //元素个数
   int *array;

   cin>>arraysize;

   array=new int[arraysize];//开辟堆内存

   for(int i=0;i<arraysize;i++)
   {
      array[i]=i;
   }

   for(int i=0;i<arraysize;i++)
   {
      cout<<array[i]<<",";
   }
   cout<<endl;
   delete[] array;//释放堆内存
   cin.get();
   cin.get();
}
```

13.C/C++中字符串常量的不相等性及字符串的 Copy

```
#include <iostream>

void main(void)
{
   if("test"=="test")
   {
      cout<<"相等";
   }
   else
   {
      cout<<"不相等";
   }
}
```

上面的代码我们测试两个内容为 test 的字符串常量是否相等，按照常理，应该是相等的，这些在一些过程式语言中会得到相等的结论，但在 c/c++却不是这样。

为什么呢？

答案在这里：因为字符串常量存储在计算机内存中，两个字符串常量的地址均不相同，所以这样的比较自然就不会得到我们所需要的结果，如果要进行是否相等的比较应该使用 strcmp()这个涵数进行比较！

```
#include <iostream>
#include <string>
using namespace std;
void main(void)
{
   if(strcmp("test","test")==0)
   {
      cout<<"相等";
   }
   else
   {
      cout<<"不相等";
   }
   cin.get();
```

```
}
```

strcmp()的函数原形是，int strcmp(const char* str1,const char* str)

相当将会返回一个等于 0 的整数，不相等的时候将会返回一个非 0 整数。

```
#include <iostream>
#include <string>
using namespace std;
void main(void)
{
   char test[]="test str!";
   char str[15];
   strcpy(str,test);
   cout<<str<<endl;

   int a[]={1,2,3,4,5};
   int b[5];
   memcpy(b,a,sizeof(a));
   for(int i=0;i<5;i++)
   {
      cout<<b[i]<<",";
   }
   cin.get();
}
```

上面的代码中的 strcpy 用来处理字符串数学组的 copy，由于字符串数组属于 const char*也就是常量指针所以是不能用 a="test str!";的方式赋值的，接在后面的 memcpy 用于处理非\0 结尾的数组的 copy 处理，memcpy 第三个参数是设置 b 在内存中所需要的内存空间大小所以用 sizeof(a)*sizeof(int)来处理。

14.C/C++中字符指针数组及指向指针的指针的含义

就指向指针的指针，很早以前在说指针的时候说过，但后来发现很多人还是比较难以理解，这一次我们再次仔细说一说指向指针的指针。

先看下面的代码，注意看代码中的注解：

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

void print_char(char* array[],int len);//函数原形声明

void main(void)
{
//-----------------------------段 1----------------------------------------
   char *a[]={"abc","cde","fgh"};//字符指针数组
   char* *b=a;//定义一个指向指针的指针,并赋予指针数组首地址所指向的第一个字符串的地址也就是 abc\0 字符串的首地址
   cout<<*b<<"|"<<*(b+1)<<"|"<<*(b+2)<<endl;
//-------------------------------------------------------------------------

//-----------------------------段 2----------------------------------------
   char* test[]={"abc","cde","fgh"};//注意这里是引号,表示是字符串,以后的地址每加 1 就是加 4 位(在 32 位系统上)
   int num=sizeof(test)/sizeof(char*);//计算字符串个数
   print_char(test,num);
   cin.get();
//-------------------------------------------------------------------------
}

void print_char(char* array[],int len)//当调用的时候传递进来的不是数组,而是字符指针他每加 1 也就是加上 sizeof(char*)的长度
{
   for(int i=0;i<len;i++)
   {
      cout<<*array++<<endl;
   }
}
```

下面我们来仔细说明一下字符指针数组和指向指针的指针，段 1 中的程序是下面的样子：

```
char *a[]={"abc","cde","fgh"};
```

```
char* *b=a;
cout<<*b<<"|"<<*(b+1)<<"|"<<*(b+2)<<endl;
```

char *a[]定义了一个指针数组,注意不是 char[], char[]是不能同时初始化为三个字符的,定义以后的 a[]其实内部有三个内存位置,分别存储了 abc\0, cde\0, fgh\0, 三个字符串的起始地址,而这三个位置的内存地址却不是这三个字符串的起始地址,在这个例子中 a[]是存储在栈空间内的,而三个字符串却是存储在静态内存空间内的 const 区域中的,接下去我们看到了 char* *b=a;这里是定义了一个指向指针的指针,如果你写成 char *b=a;那么是错误的,因为编译器会返回一个无法将 char* *[3]转换给 char *的错误,b=a 的赋值,实际上是把 a 的首地址赋给了 b,由于 b 是一个指向指针的指针,程序的输出 cout<<*b<<"|"<<*(b+1)<<"|"<<*(b+2)<<endl;

结果是

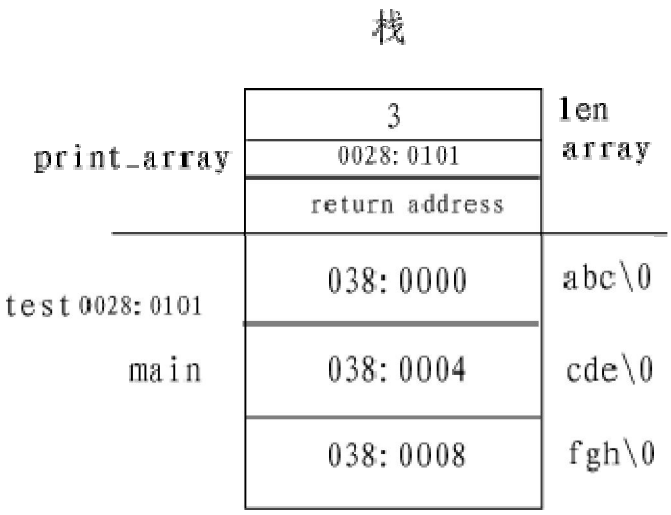
```
abc
cde
fgh
```

可以看出每一次内存地址的+1 操作事实上是一次加 sizeof(char*)的操作,我们在 32 位的系统中 sizeof(char*)的长度是 4,所以每加 1 也就是+4,实际上是*a[]内部三个位置的+1,所以*(b+1)的结果自然就是 cde 了,我们这时候可能会问,为什么输出是 cde 而不是 c 一个呢?答案是这样的,在 c++中,输出字符指针就是输出字符串,程序会自动在遇到\0 后停止.

我们最后分析一下段 2 中的代码,段 2 中我们调用了 print_array()这个函数,这个函数中形式参数是 char *array[]和代码中的 char *test[]一样,同为字符指针,当你把参数传递过来的时候,事实上不是把数组内容传递过来,test 的首地址传递了进来,由于 array 是指针,所以在内存中它在栈区,具有变量一样的性质,可以为左值,所以我们输出写成了,cout<<*array++<<endl;当然我们也可以改写为 cout<<array[i]<<endl,这里在循环中的每次加 1 操作和段 1 代码总的道理是一样的,注意看下面的图!

到这里这两个非常重要的知识点我们都说完了,说归说,要想透彻理解希望读者多动手,多观察,熟能生巧。

下面是内存结构示意图:



函数存放在内存的代码区域内，它们同样有地址，我们如何能获得函数的地址呢？

如果我们有一个 int test(int a)的函数，那么，它的地址就是函数的名字，这一点如同数组一样，数组的名字就是数组的起始地址。

定义一个指向函数的指针用如下的形式，以上面的 test()为例：

int (*fp)(int a);//这里就定义了一个指向函数的指针

函数指针不能绝对不能指向不同类型，或者是带不同形参的函数，在定义函数指针的时候我们很容易犯如下的错误。

int *fp(int a);//这里是错误的，因为按照结合性和优先级来看就是先和()结合，然后变成了一个返回整形指针的函数了，而不是函数指针，这一点尤其需要注意！

下面我们来看一个具体的例子：

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

int test(int a);

void main(int argc,char* argv[])
{
   cout<<test<<endl;//显示函数地址
   int (*fp)(int a);
   fp=test;//将函数 test 的地址赋给函数学指针 fp
   cout<<fp(5)<<"|"<<(*fp)(10)<<endl;
//上面的输出 fp(5),这是标准 c++的写法,(*fp)(10)这是兼容 c 语言的标准写法,两种同意,但注意区分,避免写的程序产生移植性问题!
   cin.get();
}

int test(int a)
{
   return a;
}
```

typedef 定义可以简化函数指针的定义，在定义一个的时候感觉不出来，但定义多了就知道方便了，上面的代码改写成如下的形式：

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

int test(int a);

void main(int argc,char* argv[])
{
```

```
    cout<<test<<endl;
    typedef int (*fp)(int a);//注意,这里不是生命函数指针,而是定义一个函数指针的类型,这个类型是自己定义的,类型名为 fp
    fp fpi;//这里利用自己定义的类型名 fp 定义了一个 fpi 的函数指针!
    fpi=test;
    cout<<fpi(5)<<"|"<<(*fpi)(10)<<endl;
    cin.get();
}

int test(int a)
{
    return a;
}
```

15.对 C++中引用的补充说明(实例)

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

void main(int argc,char* argv[])
{
    int a=10;
    int b=20;
    int &rn=a;
    cout<<rn<<"|"<<a<<endl;
    cout<<&rn<<"|"<<&a<<endl;//c++中是无法取得应用的内存地址的,取引用的地址就是取目标的地址!
    rn=b;//把引用指向另一个目标----变量 b
    cout<<&rn<<"|"<<&a<<"|"<<&b<<endl;
    rn=100;//试图改变 b 的值
    cout<<a<<"|"<<b<<endl;//输出修改后的结果
    cin.get();
}
```

由于引用本身就是目标的一个别名，引用本身的地址是一个没有意义的值，所以在 c++中是无法取得引用的内存地址的。取引用的地址就是取目标的地址，c++本身就根本不提供获取引用内存地址的方法。

引用一单初始化,就不在能够被指向其它的目标，虽然编译不会出错，但操作是不起作用的，实际上还是指向最先指向的目标。

上面代码中的 rn=b 实际在计算机看来就是 a=b，所以修改的还是 a 的值。

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

void main(int argc,char* argv[])
{
    int a=10;
    void &rn=a;// 错误的,void 即无类型的类型
    int a[100];
    int &ra[100]=a;//错误,不能声明引用数组
    cin.get();
```

```
}
```

上面的两错误要记住引用的特性，void 修饰是不能够声明引用的，引用是不能够声明数组的，即不能够声明引用数组。

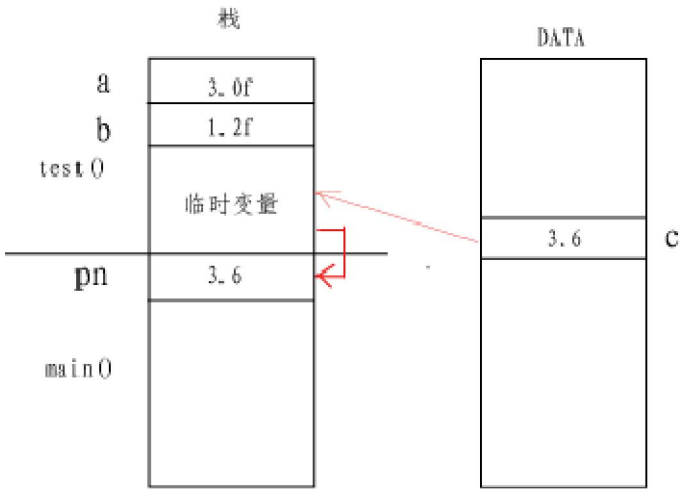
下面我们要说一下，也是补充中最重要最需要掌握的内容，也是对传统函数操作的内存状态的一个补充学习。

下面我们来看一个例子：

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

float c;
float test(float,float);
void main(int argc,char* argv[])
{
   float pn=test(3.0f,1.2f);
   cout<<pn;
   cin.get();
}

float test(float a,float b)
{
   c=a*b;
   return c;
}
```

在上面的代码中我们可能我们可能以为函数返回的就是 c 变量，呵呵。这么想可能就错了，普通情况下我们在函数内进行普通值返回的时候在内存栈空间内其实是自动产生了一个临时变量 temp，它是返回值的一个副本一个 copy，函数在 return 的时候其实是 return 的这个临时产生的副本。

数据在内存中的情况如下图：



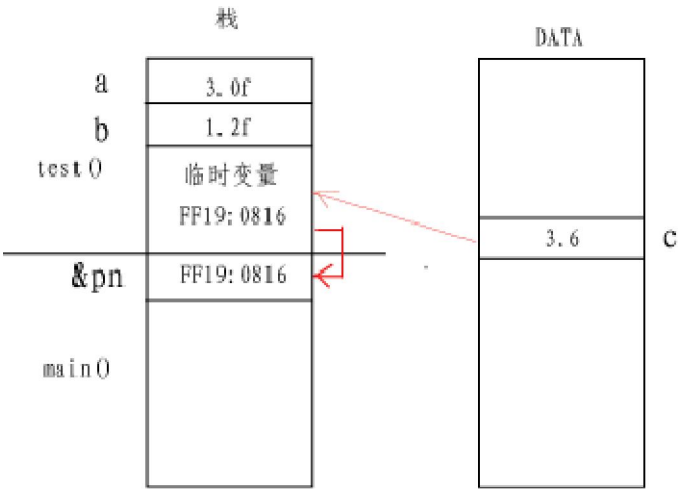
上图明确表示了副本领事变量的情况。

下面我们再来看一种情况,就是把返回值赋给引用:

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

float c;
float test(float,float);
void main(int argc,char* argv[])
{
   float &pn=test(3.0f,1.2f);//警告:返回的将是临时变量,pn 引用将成为临时变量的别名!
   cout<<pn;
   cin.get();
}

float test(float a,float b)
{
   c=a*b;
   return c;
}
```

float &pn=test(3.0f,1.2f);这句在 bc 中能够编译通过，因为 bc 扩展设置为临时变量设置引用，那么临时变量的生命周期将和引用的生命周期一致，但在 vc 中却不能通过编译，因为一但 test()执行过后临时变量消失在栈空间内，这时候 pn 将成为一个没有明确目标的引用，严重的时候会导致内存出错。

它在内存中的情况见下图:



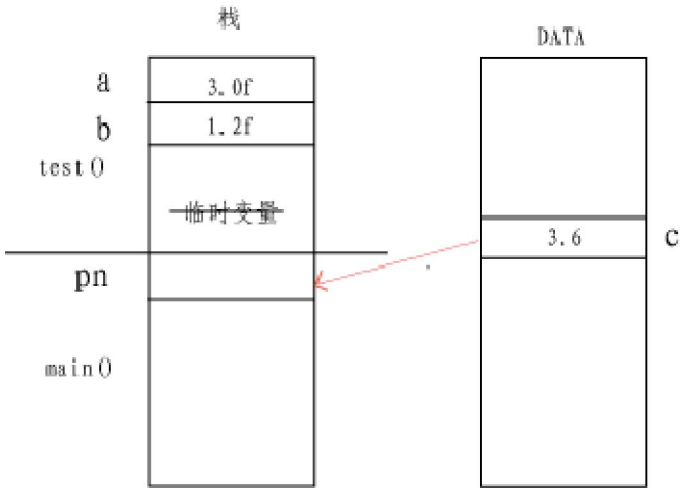
我们在图中看到，由于函数仍然是普通方法返回，所以仍然会有一个副本临时变量产生，只不过，这一次只是返回一个目标地址，在 main 中目标地址被赋予了引用 pn。

下面我们再看一种情况，这是返回引用给变量的情况：

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

float c;
float& test(float,float);
void main(int argc,char* argv[])
{
   float pn=test(3.0f,1.2f);
   cout<<pn;
   cin.get();
}

float &test(float a,float b)
{
   c=a*b;
   return c;
}
```

这种返回引用给变量的情况下，在内存中，test()所在的栈空间内并没有产生临时变量，而是直接将全局变量 c 的值给了变量 pn，这种方式是我们最为推荐的操作方式，因为不产生临时变量直接赋值的方式可以节省内存空间提高效率，程序的可读性也是比较好的。

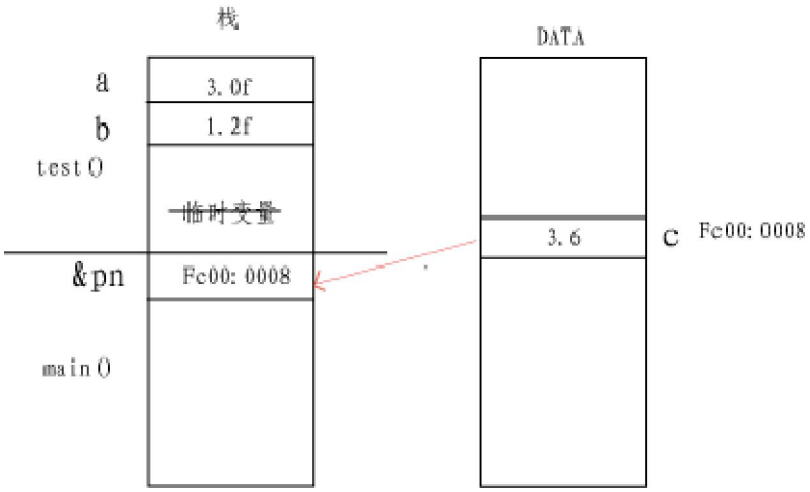
它在内存中的情况见下图：



最后的一种情况是函数返回引用，并且发值赋给一个引用的情况：

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

float c;
float& test(float,float);
void main(int argc,char* argv[])
{
   float &pn=test(3.0f,1.2f);
   cout<<pn;
   cin.get();
}

float &test(float a,float b)
{
   c=a*b;
   return c;
}
```

这种情况同样也不产生临时变量，可读和性能都很好，但有一点容易弄错，就是当 c 是非 main 的局部变量或者是在堆内存中临时开辟后来又被 fee 掉了以后的区域，这种情况和返回的指针是局部指针的后果一样严重，会导致引用指向了一个不明确的地址，这种情况在内存中情况见下图：



由于这种情况存在作用域的问题，故我们推荐采用第三种方式处理。

接下来我们说几个利用引用作为左值参与计算的例子，这一点一非常重要，对于理解返回引用的函数是非常有帮助的。

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

float c;
float& test(float,float);
void main(int argc,char* argv[])
{
   float &pn=test(3.0f,1.2f);
   cout<<pn<<endl;
   test(3.0f,1.2f)=12.1;//把函数作左值进行计算!
   cout<<pn;
   cin.get();
}

float &test(float a,float b)
{
   c=a*b;
   return c;
}
```

通常来说函数是不能作为左值，因为引用可以做为左值，所以返回引用的函数自然也就可以作为左值来计算了。

在上面的代码中：

```
float &pn=test(3.0f,1.2f);
```

进行到这里的时候 pn 已经指向到了目标 c 的地址了。

接下来运行了

```
test(3.0f,1.2f)=12.1;
```

把函数作左值进行计算，这里由于 test 是返回引用的函数，其实返回值返回的地址就是 c 的地址，自然 c 的值就被修改成了 12.1。

16.新手入门：C/C++中的结构体

什么是结构体?

简单的来说，结构体就是一个可以包含不同数据类型的一个结构，它是一种可以自己定义的数据类型，它的特点和数组主要有两点不同，首先结构体可以在一个结构中声明不同的数据类型，第二相同结构的结构体变量是可以相互赋值的，而数组是做不到的，因为数组是单一数据类型的数据集合，它本身不是数据类型(而结构体是)，数组名称是常量指针，所以不可以做为左值进行运算，所以数组之间就不能通过数组名称相互复制了，即使数据类型和数组大小完全相同。

定义结构体使用 struct 修饰符，例如：

```
struct test
{
float a;
int b;
};
```

上面的代码就定义了一个名为 test 的结构体，它的数据类型就是 test，它包含两个成员 a 和 b，成员 a 的数据类型为浮点型，成员 b 的数据类型为整型。

由于结构体本身就是自定义的数据类型，定义结构体变量的方法和定义普通变量的方法一样。

```
test pn1;
```

这样就定义了一 test 结构体数据类型的结构体变量 pn1，结构体成员的访问通过点操作符进行，pn1.a=10 就对结构体变量 pn1 的成员 a 进行了赋值操作。

注意:结构体生命的时候本身不占用任何内存空间，只有当你用你定义的结构体类型定义结构体变量的时候计算机才会分配内存。

结构体，同样是可以定义指针的，那么结构体指针就叫做结构指针。

结构指针通过->符号来访问成员，下面我们就以上所说的看一个完整的例子：

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

struct test//定义一个名为 test 的结构体
{
```

```
    int a;//定义结构体成员 a
    int b;//定义结构体成员 b
};

void main()
{
    test pn1;//定义结构体变量 pn1
    test pn2;//定义结构体变量 pn2

    pn2.a=10;//通过成员操作符.给结构体变量 pn2 中的成员 a 赋值
    pn2.b=3;//通过成员操作符.给结构体变量 pn2 中的成员 b 赋值

    pn1=pn2;//把 pn2 中所有的成员值复制给具有相同结构的结构体变量 pn1
    cout<<pn1.a<<"|"<<pn1.b<<endl;
    cout<<pn2.a<<"|"<<pn2.b<<endl;

    test *point;//定义结构指针

    point=&pn2;//指针指向结构体变量 pn2 的内存地址
    cout<<pn2.a<<"|"<<pn2.b<<endl;
    point->a=99;//通过结构指针修改结构体变量 pn2 成员 a 的值
    cout<<pn2.a<<"|"<<pn2.b<<endl;
    cout<<point->a<<"|"<<point->b<<endl;
    cin.get();
}
```

总之，结构体可以描述数组不能够清晰描述的结构，它具有数组所不具备的一些功能特性。

下面我们来看一下，结构体变量是如何作为函数参数进行传递的。

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

struct test
{
    char name[10];
    float socre;
};

void print_score(test pn)//以结构变量进行传递
{
    cout<<pn.name<<"|"<<pn.socre<<endl;
}
```

```
void print_score(test *pn)//一结构指针作为形参
{
   cout<<pn->name<<"|"<<pn->socre<<endl;
}

void main()
{
   test a[2]={{"marry",88.5},{"jarck",98.5}};
   int num = sizeof(a)/sizeof(test);
   for(int i=0;i<num;i++)
   {
      print_score(a[i]);
   }
   for(int i=0;i<num;i++)
   {
      print_score(&a[i]);
   }
   cin.get();
}
```

void print_score(test *pn)的效率是要高过 void print_score(test pn)的，因为直接内存操作避免了栈空间开辟结构变量空间需求，节省内存。

下面我们再说一下，传递结构引用的例子。

利用引用传递的好处很多，它的效率和指针相差无几，但引用的操作方式和值传递几乎一样，种种优势都说明善用引用可以做到程序的易读和易操作，它的优势尤其在结构和大的时候，避免传递结构变量很大的值，节省内存，提高效率。

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

struct test
{
   char name[10];
   float socre;
};

void print_score(test &pn)//以结构变量进行传递
{
   cout<<pn.name<<"|"<<pn.socre<<endl;
}
```

```
void main()
{
   test a[2]={{"marry",88.5},{"jarck",98.5}};
   int num = sizeof(a)/sizeof(test);
   for(int i=0;i<num;i++)
   {
      print_score(a[i]);
   }
   cin.get();
}
```

上面我们说明了易用引用对结构体进行操作的优势，下面我们重点对比两个例程，进一部分析关于效率的问题。

```
//-------------------------------------例程 1--------------------------------

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

struct test
{
   char name[10];
   float socre;
};

void print_score(test &pn)
{
   cout<<pn.name<<"|"<<pn.socre<<endl;
}

test get_score()
{
   test pn;
   cin>>pn.name>>pn.socre;
   return pn;
}
void main()
{
   test a[2];
   int num = sizeof(a)/sizeof(test);
   for(int i=0;i<num;i++)
   {
      a[i]=get_score();
   }
```

```
    cin.get();
    for(int i=0;i<num;i++)
    {
        print_score(a[i]);
    }
    cin.get();
}

//-------------------------------------例程 2---------------------------------

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

struct test
{
    char name[10];
    float socre;
};

void print_score(test &pn)
{
    cout<<pn.name<<"|"<<pn.socre<<endl;
}

void get_score(test &pn)
{
    cin>>pn.name>>pn.socre;
}
void main()
{
    test a[2];
    int num = sizeof(a)/sizeof(test);
    for(int i=0;i<num;i++)
    {
        get_score(a[i]);
    }
    cin.get();
    for(int i=0;i<num;i++)
    {
        print_score(a[i]);
    }
    cin.get();
}
```

例程 2 的效率要远高过例程 1 的原因主要有以下两处：

第一：

例程 1 中的

```
test get_score()
{
test pn;
cin>>pn.name>>pn.socre;
return pn;
}
```

调用的时候在内部要在栈空间开辟一个名为 pn 的结构体变量，程序 pn 的时候又再次在栈内存空间内自动生成了一个临时结构体变量 temp，在前面的教程中我们已经说过，它是一个 copy，而例程 2 中的：

```
void get_score(test &pn)
{
cin>>pn.name>>pn.socre;
}
```

却没有这一过程，不开辟任何新的内存空间，也没有任何临时变量的生成。

第二：

例程 1 在 mian()中，必须对返回的结构体变量进行一次结构体变量与结构体变量直接的相互赋值操作。

```
for(int i=0;i<num;i++)
{
a[i]=get_score();
}
```

而例程 2 中由于是通过内存地址直接操作，所以完全没有这一过程，提高了效率。

```
for(int i=0;i<num;i++)
{
get_score(a[i]);
}
```

函数也是可以返回结构体应用的，例子如下：

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

```
struct test
{
   char name[10];
   float socre;
};

test a;



test &get_score(test &pn)
{
   cin>>pn.name>>pn.socre;
   return pn;
}

void print_score(test &pn)
{
   cout<<pn.name<<"|"<<pn.socre<<endl;
}

void main()
{
   test &sp=get_score(a);
   cin.get();
   cout<<sp.name<<"|"<<sp.socre;
   cin.get();
}
```

调用 get_score(a);结束并返回的时候，函数内部没有临时变量的产生，返回直接吧全局结构变量 a 的内存地址赋予结构引用 sp

最后提一下指针的引用

定义指针的引用方法如下：

```
void main()
{
int a=0;
int b=10;
int *p1=&a;
int *p2=&b;
int *&pn=p1;
cout <<pn<<"|"<<*pn<<endl;
pn=p2;
cout <<pn<<"|"<<*pn<<endl;
```

```
cin.get();
}
```

pn 就是一个指向指针的引用，它也可以看做是指针别名，总之使用引用要特别注意它的特性，它的操作是和普通指针一样的，在函数中对全局指针的引用操作要十分小心，避免破坏全局指针！

17.C/C++中结构体(struct)知识点强化

在上一个教程中我们已经简单的阐述了什么是结构体了，为了进一部的学习结构体这一重要的知识点，我们今天来学习一下链表结构。

结构体可以看做是一种自定义的数据类型，它还有一个很重要的特性，就是结构体可以相互嵌套使用，但也是有条件的，结构体可以包含结构体指针，但绝对不能在结构体中包含结构体变量。

```
struct test
{
    char name[10];
    float socre;
    test *next;
};//这样是正确的!
struct test
{
    char name[10];
    float socre;
    test next;
};//这样是错误的!
```

利用结构体的这点特殊特性，我们就可以自己生成一个环环相套的一种射线结构，一个指向另一个。

链表的学习不像想象的那么那么容易，很多人学习到这里的时候都会碰到困难，很多人也因此而放弃了学习，在这里我说,一定不能放弃，对应它的学习我们要进行分解式学习，方法很重要，理解需要时间，不必要把自己逼迫的那么紧，学习前你也得做一些最基本的准备工作，你必须具备对堆内存的基本知识的了解，还有就是对结构体的基本认识，有了这两个重要的条件，再进行分解式学习就可以比较轻松的掌握这一节内容的难点。

下面我们给出一个完整的创建链表的程序，不管看的懂看不懂希望读者先认真看一下，想一想，看不懂没有关系，因为我下面会有分解式的教程，但之前的基本思考一定要做，要不即使我分解了你也是无从理解的。

代码如下，我在重要部分做了注解：

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

struct test
{
    char name[10];
    float socre;
```

```
    test *next;
};

test *head;//创建一个全局的引导进入链表的指针

test *create()
{
    test *ls;//节点指针
    test *le;//链尾指针
    ls = new test;//把 ls 指向动态开辟的堆内存地址
    cin>>ls->name>>ls->socre;
    head=NULL;//进入的时候先不设置 head 指针指向任何地址,因为不知道是否一上来就输入 null 跳出程序
    le=ls;//把链尾指针设置成刚刚动态开辟的堆内存地址,用于等下设置 le->next,也就是下一个节点的位置

    while(strcmp(ls->name,"null")!=0)//创建循环条件为 ls->name 的值不是 null,用于循环添加节点
    {
        if(head==NULL)//判断是否是第一次进入循环
        {
            head=ls;//如果是第一次进入循环,那么把引导进入链表的指针指向第一次动态开辟的堆内存地址
        }
        else
        {
            le->next=ls;//如果不是第一次进入那么就把上一次的链尾指针的 le->next 指向上一次循环结束前动态创建的堆内存地址
        }
        le=ls;//设置链尾指针为当前循环中的节点指针,用于下一次进入循环的时候把上一次的节点的 next 指向上一次循环结束前动态创建
的堆内存地址
        ls=new test;//为下一个节点在堆内存中动态开辟空间
        cin>>ls->name>>ls->socre;
    }

    le->next=NULL;//把链尾指针的 next 设置为空,因为不管如何循环总是要结束的,设置为空才能够在循环显链表的时候不至于死循环
    delete ls;//当结束的时候最后一个动态开辟的内存是无效的,所以必须清除掉
    return head;//返回链首指针
}

void showl(test *head)
{
    cout<<"链首指针:"<<head<<endl;
    while(head)//以内存指向为 null 为条件循环显示先前输入的内容
    {
        cout<<head->name<<"|"<<head->socre<<endl;
        head=head->next;
    }
}
```

```
void main()
{
   showl(create());
   cin.get();
   cin.get();
}
```

上面的代码我们是要达到一个目的：就是要存储你输入的人名和他们的得分，并且以链状结构把它们组合成一个链状结构。

程序种有两个组成部分

```
test *create()
```

和

```
void showl(test *head)
```

这两个函数，create 是用来创建链表的 ，showl 是用来显示链表的。

create 函数的返回类型是一个结构体指针，在程序调用的时候我们用了 showl(create());，而不用引用的目的原因是引导指针是一个全局指针变量，我们不能在 showl()内改变它，因为 showl()函数内有一个移动操作 head=head->next;，如果是引用的话我们就破坏了 head 指针的位置，以至于我们再也无法找会首地址的位置了。

下面我们来分解整个程序，以一个初学者的思想来思考整个程序，由浅入深的逐步解释。

首先，我们写这个程序，要考虑到由于是一个链表结构，我们不可能知道它的大小到底是多大，这个问题我们可以用动态开辟堆内存来解决，因为堆内存在程序结束前始终是有效的，不受函数栈空间生命期的限制，但要注意的是我们必须有一个指针变量来存储这一链状结构的进入地址，而在函数内部来建立这一指针变量显然是不合适的，因为函数一旦退出，这个指针变量也随之失效，所以我们在程序的开始声明了一个全局指针变量。

```
test *head;//创建一个全局的引导进入链表的指针
```

好解决了这两个问题，我们接下去思考

有输入就必然有输出，由于输出函数和输入函数是相对独立的，为了不断测试程序的正确性好调试我们先写好输出函数和 main 函数捏的调用，创建函数我们先约定好名为 create。

我们先写出如下的代码：

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

struct test
{
```

```
    char name[10];
    float socre;
    test *next;
};

test *head;//创建一个全局的引导进入链表的指针

test *create()
{

    return head;//返回链首指针
}

void showl(test *head)
{
    cout<<"链首指针:"<<head<<endl;
    while(head)//以内存指向为 null 为条件循环显示先前输入的内容
    {
        cout<<head->name<<"|"<<head->socre<<endl;
        head=head->next;
    }
}

void main()
{
    showl(create());
    cin.get();
    cin.get();
}
```

程序写到这里，基本形态已经出来，输入和调用我们已经有了。

下面我们来解决输入问题，链表的实现我们是通过循环输入来实现的，既然是循环我们就一定得考虑终止循环的条件，避免死循环和无效循环的发生。

在 create()函数内部我们先写成这样：

```
test *create()
{
    test *ls;//节点指针
    test *le;//链尾指针
    ls = new test;//把 ls 指向动态开辟的堆内存地址
    cin>>ls->name>>ls->socre;
    head=NULL;//进入的时候先不设置 head 指针指向任何地址,因为不知道是否一上来就输入 null 跳出程序
```

```
    le=ls;//把链尾指针设置成刚刚动态开辟的堆内存地址,用于等下设置 le->next,也就是下一个节点的位置


    le->next=NULL;//把链尾指针的 next 设置为空,因为不管如何循环总是要结束的,设置为空才能够在循环显链表的时候不至于死循环
    delete ls;//当结束的时候最后一个动态开辟的内存是无效的,所以必须清除掉
    return head;//返回链首指针
}
```

在循环创建之前我们必须考虑一个都不输入的情况。

程序一单进入 create 函数我们首先必然要创建一个节点，我们先创建一个节点指针，后把者个节点指针指向到动态开辟的 test 类型的动态内存地址位置上。

所以我们有了

```
test *ls;
ls = new test;
```

程序既然是循环输入，而结构成员 test *next 又是用来存储下一个接点的内存地址的，每次循环我们又要动态创建一个新的内存空间，所以我们必须要有一个指针来存储上一次循环动态开辟的内存地址，于是就有了

```
test *le;
```

接下来在进入循环前我们要创建链表的第一个节点，第一个节点必然是在循环外创建，于是就有了

```
cin>>ls->name>>ls->socre;
```

程序执行者的情况是位置的，所以我们必然要考虑，一上来就不想继续运行程序的情况，所以我们一开始先把 head 引导指针设置为不指向任何地址也就是

```
head=NULL;
```

为了符合 le 也就是链尾指针的设计思路，我们在循环前一定要保存刚刚动态开辟的内存地址，好在下一次循环的时候设置上一个节点中的 next 成员指向，于是我们便有了：

```
le=ls;
```

为了实现循环输入我们又了下面的代码：

```
while(strcmp(ls->name,"null")!=0)
{
if(head==NULL)
```

```
{
head=ls;
}
else
{
le->next=ls;
}
le=ls;
ls=new test;
cin>>ls->name>>ls->socre;
}
```

程序是循环必然要有终止循环的条件，所以我们的循环条件是：

```
while(strcmp(ls->name,"null")!=0)
```

输入的名字是 null 的时候就停止循环。

为了保证第一次进入循环，也就是在循环内准备创建第二个节点前，设置引导指针的指向我们有了如下的判断代码：

```
if(head==NULL)
{
head=ls;
}
else
{
le->next=ls;
}
```

代码中的 else 条件是为了设置前一个节点 next 指向而写的,这点我们记住先看下面的代码,稍后大家回过头想就明白了

```
le=ls;
ls=new test;
cin>>ls->name>>ls->socre;
```

le=ls;这么写就是为了保存上一次循环指针的位置而设的,正是为了上面的 else 代码而做的预先保留

```
ls=new test;
cin>>ls->name>>ls->socre;
```

这两行代码的意思就是继续开辟下一个节点空间,和输入节点内容!

循环一旦结束也就结束了程序,为了保持程序不出错,也就是最后一个节点的 next 成员指向为空我们有了下面的代码

```
le->next=NULL;
```

程序的思路始终是以先开辟后判断为思路的,所以到最后一个不成立的时候总会有一个多开辟的内存空间,为了删除掉它,我们有了下面的代码

```
delete ls;
```

程序到最后由于返回 head 指针

```
return head;
```

显示链表的函数没有什么太多特别的也只需要注意下面这样就可以了!

```
head=head->next;
```

我们之所以不用 head+=1;来写就是因为链表是我们动态开辟的，而每一个节点的位置并不是相连的，next 成员指针的意义也就是下一个节点的内存地址。

到这里整个创建函数的设计思路也都说完了，笔者不一定说的很好，但基本思路是这样的，希望读者多思考，多对比，相信此教程还是对大家有帮助的，程序设计就是利用逐步思考的方式进行的，写好的代码往往直接看看不懂就是因为中间的细节并不是一次都能够想到的。

下面我们来说一下链表节点的删除!

我们以上面的程序为基础，但为了我们方便学习删除我们休整结构体为

```
struct test
{
int number;
float socre;
test *next;
};
```

number 为唯一的编号每一个节点的。

删除的我就不多说了，里面重要部分有注解。

特别注意 deletel 函数的参数意义，指针的引用在这里很重要，如果只是指针，或者只是应用都是不行的，为什么仔细思考，很多知名的教材在这一问题上都很模糊，而且很多书还有错误，程序不错，但思路是错的，我这里特别不说，请大家仔细阅读程序，如果还是有问题，可以回此帖，我会回答的。

完整代码如下：

```
#include <iostream>
using namespace std;
struct test
{
   int number;
   float socre;
   test *next;
};
test *head;//创建一个全局的引导进入链表的指针

test *create()
{
   test *ls;//节点指针
   test *le;//链尾指针
   ls = new test;//把 ls 指向动态开辟的堆内存地址
   cin>>ls->number>>ls->socre;
   head=NULL;//进入的时候先不设置 head 指针指向任何地址,因为不知道是否一上来就输入 null 跳出程序
   le=ls;//把链尾指针设置成刚刚动态开辟的堆内存地址,用于等下设置 le->next,也就是下一个节点的位置
   while(ls->number!=0)//创建循环条件为 ls->number 的值不是 null,用于循环添加节点
   {
      if(head==NULL)//判断是否是第一次进入循环
      {
         head=ls;//如果是第一次进入循环,那么把引导进入链表的指针指向第一次动态开辟的堆内存地址
      }
      else
      {
         le->next=ls;//如果不是第一次进入那么就把上一次的链尾指针的 le->next 指向上一次循环结束前动态创建的堆内存地址
      }
      le=ls;//设置链尾指针为当前循环中的节点指针,用于下一次进入循环的时候把上一次的节点的 next 指向上一次循环结束前动态创建
的堆内存地址
      ls=new test;//为下一个节点在堆内存中动态开辟空间
      cin>>ls->number>>ls->socre;
   }
   le->next=NULL;//把链尾指针的 next 设置为空,因为不管如何循环总是要结束的,设置为空才能够在循环显链表的时候不至于死循环
   delete ls;//当结束的时候最后一个动态开辟的内存是无效的,所以必须清除掉
   return head;//返回链首指针
}
void showl(test *head)
{
   cout<<"链首指针:"<<head<<endl;
   while(head)//以内存指向为 null 为条件循环显示先前输入的内容
   {
```

```
        cout<<head->number<<"|"<<head->socre<<endl;
        head=head->next;
    }
}
void deletel(test *&head,int number)//这里如果参数换成 test *head,意义就完全不同了,head 变成了复制而不是原有链上操作了,特别注意,很多书上都不对这里
{
    test *point;//判断链表是否为空
    if(head==NULL)
    {
        cout<<"链表为空,不能进行删除工作!";
        return;
    }
    if(head->number==number)//判删除的节点是否为首节点
    {
        point=head;
        cout<<"删除点是链表第一个节点位置!";
        head=head->next;//重新设置引导指针
        delete point;
        return;
    }
    test *fp=head;//保存连首指针
    for(test *&mp=head;mp->next;mp=mp->next)
    {
        if(mp->next->number==number)
        {
            point=mp->next;
            mp->next=point->next;
            delete point;
            head=fp;//由于 head 的不断移动丢失了 head,把进入循环前的 head 指针恢复!
            return;
        }
    }
}
void main()
{
    head=create();//调用创建
    showl(head);
    int dp;
    cin>>dp;
    deletel(head,dp);//调用删除
    showl(head);
    cin.get();
    cin.get();
```

}

最后我学习一下如何在已有的链表上插入节点

我们要考虑四中情况,

1. 链表为空!

2. 插入点在首节点前

3. 插入点找不到的情况我们设置放在最后!

4. 插入点在中间的情况!

今天的程序在昨天的基础上做了进一步的修改,可以避免删除点找不到的情况,如果找不到删除点就退出函数!

```
#include <iostream>
using namespace std;
struct test
{
   int number;
   float socre;
   test *next;
};
test *head;//创建一个全局的引导进入链表的指针

test *create()
{
   test *ls;//节点指针
   test *le;//链尾指针
   ls = new test;//把 ls 指向动态开辟的堆内存地址
   cout<<"请输入第一个节点 number 和节点 score,输入 0.0 跳出函数"<<endl;
   cin>>ls->number>>ls->socre;
   head=NULL;//进入的时候先不设置 head 指针指向任何地址,因为不知道是否一上来就输入 null 跳出程序
   le=ls;//把链尾指针设置成刚刚动态开辟的堆内存地址,用于等下设置 le->next,也就是下一个节点的位置
   while(ls->number!=0)//创建循环条件为 ls->number 的值不是 null,用于循环添加节点
   {
      if(head==NULL)//判断是否是第一次进入循环
      {
         head=ls;//如果是第一次进入循环,那么把引导进入链表的指针指向第一次动态开辟的堆内存地址
      }
      else
      {
         le->next=ls;//如果不是第一次进入那么就把上一次的链尾指针的 le->next 指向上一次循环结束前动态创建的堆内存地址
```

```
    }
    le=ls;//设置链尾指针为当前循环中的节点指针,用于下一次进入循环的时候把上一次的节点的 next 指向上一次循环结束前动态创建的堆内存地址
    ls=new test;//为下一个节点在堆内存中动态开辟空间
    cout<<"请下一个节点 number 和节点 score,输入 0.0 跳出函数"<<endl;
    cin>>ls->number>>ls->socre;
  }
  le->next=NULL;//把链尾指针的 next 设置为空,因为不管如何循环总是要结束的,设置为空才能够在循环显链表的时候不至于死循环
  delete ls;//当结束的时候最后一个动态开辟的内存是无效的,所以必须清除掉
  return head;//返回链首指针
}
void showl(test *head)
{
  cout<<"链首指针:"<<head<<endl;
  while(head)//以内存指向为 null 为条件循环显示先前输入的内容
  {
    cout<<head->number<<"|"<<head->socre<<endl;
    head=head->next;
  }
}
void deletel(test *&head,int number)//这里如果参数换成 test *head,意义就完全不同了,head 变成了复制而不是原有链上操作了,特别注意,很多书上都不对这里
{
  test *point;//判断链表是否为空
  if(head==NULL)
  {
    cout<<"链表为空,不能进行删除工作!";
    return;
  }

  int derror=1;//设置找不到的情况的条件,预先设置为真
  test *check=head;
  while(check)//利用循环进行查找
  {
    if (check->number==number)
    {
      derror=0;//条件转为假
    }
    check=check->next;
  }
  if(derror)//如果为假就跳出函数
  {
    return;
  }
```

```
    if(head->number==number)//判删除的节点是否为首节点
    {
        point=head;
        cout<<"删除点是链表第一个节点位置!";
        head=head->next;//重新设置引导指针
        delete point;
        return;
    }
    test *fp=head;//保存连首指针
    for(test *&mp=head;mp->next;mp=mp->next)
    {
        if(mp->next->number==number)
        {
            point=mp->next;
            mp->next=point->next;
            delete point;
            head=fp;//由于 head 的不断移动丢失了 head,把进入循环前的 head 指针恢复!
            return;
        }
    }
}

void insterl(int number)
{
    test *point=new test;
    cout<<"请输入节点 number 和节点 score"<<endl;
    cin>>point->number>>point->socre;

    if(head==NULL)//链表为空的情况下插入
    {
        head=point;
        point->next=NULL;
        return;
    }

    int ierror=1;//设置找不到的情况的条件,预先设置为真
    test *le;
    test *check=head;
    while(check)//利用循环进行查找
    {
        if (check->number==number)
        {
            ierror=0;//条件转为假
```

```
        }
        le=check;
        check=check->next;
    }
    if(ierror)
    {
        cout<<le->number;
        le->next=point;
        point->next=NULL;
        return;
    }

   if(head->number==number)//检测是否是在第一个节点处插入
    {
        point->next=head;
        head=point;
        return;
    }

    for(test *&mp=head;mp->next;mp=mp->next)//在链表中间插入
    {
        if(mp->next->number==number)
        {
            point->next=mp->next;
            mp->next=point;
            return;
        }
    }

}
void main()
{
    head=create();//调用创建
    showl(head);
    int dp;
    cout<<"请输入删除点如果找不到就跳出函数"<<endl;
    cin>>dp;
    deletel(head,dp);//调用删除
    showl(head);
    int ip;
    cout<<"请输入插入点如果找不到就在链尾添加"<<endl;
    cin>>ip;
    insterl(ip);
    showl(head);
```

```
    cin.get();
    cin.get();
}
```

到此关于结构体的内容已经全部讨论结束，链表的建立删除插入操作可以很好的对前面所学知识进行一个总结，它既考察了程序员对内存大理解(堆内存操作、指针操作)也考察了对结构化编程掌握的熟悉程序。

以后的教程我们将着重训练面向对象的编程的相关知识点。

18.C++面向对象编程入门：类(class)

上两篇内容我们着重说了结构体相关知识的操作。

以后的内容我们将逐步完全以 c++作为主体了，这也意味着我们的教程正式进入面向对象的编程了。

前面的教程我已经再三说明，结构体的掌握非常重要，重要在哪里呢？重要在结构体和类有相同的特性，但又有很大的区别，类是构成面向对象编程的基础，但它是和结构体有着极其密切的关系。

我们在 c 语言中创建一个结构体我们使用如下方法：

```
struct test
{
   private:
      int number;
   public:
      float socre;
};
```

类的创建方式和结构体几乎一样，看如下的代码：

```
class test
{
   private:
      int number;
   public:
      float socre;
   public:
      int rp()
      {
         return number;
      }
      void setnum(int a)
      {
         number=a;
      }
};
```

但是大家注意到没有，**标准 c 中是不允许在结构体中声明函数的**，但 c++中的类可以，这一点就和 c 有了本质的区别，很好的体现了 c++面向对象的特点！

过去的 c 语言是一种非面向对象的语言

他的特性是：

程序=算法+数据结构

但 c++的特性是

对象=算法+数据结构

程序=对象+对象+对象+对象+.....

所以根据这一特性，我们在定义一个自己定义的结构体变量的时候。这个变量就应该是叫做对象或者叫实例。

例如

```
test a;
```

那么 a 就是 test 结构的一个对象(实例)

test 结构体内的成员可以叫做是分量，例如:

```
a.socre=10.1f;
```

那么 number 就是 test 结构的对象 a 的分量(或者叫数据成员,或者叫属性)score;

在 c 语言中结构体中的各成员他们的默认存储控制是 public 而 c++中类的默认存储控制是 private，所以在类中的成员如果需要外部掉用一定要加上关键字 public 声明成公有类型，这一特性同样使用于类中的成员函数，函数的操作方式和普通函数差别并不大。

例如上面的例子中的 rp()成员函数，我们如果有如下定义：

```
test a;
```

的话，调用 rp()就应该写成：

```
a.rp();
```

成员函数的调用和普通成员变量的调用方式一致都采用.的操作符。

这一小节为了巩固联系我给出一个完整的例子。

如下(重要和特殊的地方都有详细的注解)：

```
#include <iostream>
using namespace std;
class test
{
   private://私有成员类外不能够直接访问
      int number;
   public://共有成员类外能够直接访问
      float socre;
   public:
      int rp()
      {
         return number;
      }
      void setnum(int a)
      {
         number=a;
      }
};

void main()
{
   test a;
   //a.number=10;//错误的,私有成员不能外部访问
   a.socre=99.9f;
   cout<<a.socre<<endl;//公有成员可以外部访问
   a.setnum(100);//通过公有成员函数 setnum()间接对私有成员 number 进行赋值操作
   cout<<a.rp();//间接返回私有成员 number 的值
   cin.get();
}
```

好了，介绍了在类内部定义成员函数(方法)的方法，下面我们要介绍一下域区分符(::)的作用了。

下面我们来看一个例子，利用这个例子中我们要说明两个重要问题：

```
#include <iostream>
using namespace std;
int pp=0;
class test
{
   private:
      int number;
   public:
      float socre;
```

```
    int pp;
  public:
    void rp();
};
void test::rp()//在外部利用域区分符定义 test 类的成员函数
{
  ::pp=11;//变量名前加域区分符给全局变量 pp 赋值
  pp=100;//设置结构体变量
}

void main()
{
  test a;
    test b;
  a.rp();
  cout<<pp<<endl;
  cout<<a.pp<<endl;

  cin.get();
}
```

问题 1：

利用域区分符我们可以在类定义的外部设置成员函数，但要注意的是，在类的内部必须预先声明：

```
void test::rp()
```

在函数类型的后面加上类的名称再加上域区分符(::)再加函数名称，利用这样的方法我们就在类的外部建立了一个名为 rp 的 test 类大成员函数(方法)，可能很多人要问，这么做有意义吗？在类的内部写函数代码不是更好？

答案是这样的：在类的定义中，一般成员函数的规模一般都比较小，而且一些特殊的语句是不能够使用的，而且一般会被自动的设置成为 inline(内联)函数,即使你没有明确的声明为 inline，那么为什么有会被自动设置成为 inline 呢？因为大多数情况下，类的定义一般是放在头文件中的，在编译的时候这些函数的定义也随之进入头文件，这样就会导致被多次编译，如果是 inline 的情况，函数定义在调用处扩展，就避免了重复编译的问题，而且把大量的成员函数都放在类中使用起来也十分不方便，为了避免这种情况的发生，所以 c++是允许在外部定义类的成员函数(方法)的，将类定义和其它成员函数定义分开，是面向对象编程的通常做法，我们把类的定义在这里也就是头文件了看作是类的外部接口，类的成员函数的定义看成是类的内部实现。写程序的时候只需要外部接口也就是头文件即可，这一特点**和我们使用标准库函数的道理是一致**的，因为在类的定义中，已经包含了成员函数(方法)的声明。

问题二

域区分符和外部全局变量和类成员变量之间的关系。

在上面的代码中我们看到了，外部全局和类内部都有一个叫做 pp 的整形变量，那么我们要区分操作他们用什么方法呢？

使用域区分符就可以做到这一点，在上面的代码中::pp=11;操作的就是外部的同名称全局变量，pp=100;操作的就是类内部的成员变量，这一点十分重要！

问题三

一个类的所有对象调用的都是同一段代码，那么操作成员变量的时候计算机有是如何知道哪个成员是属于哪个对象的呢？

这里牵扯到一个**隐藏的 this 指针**的问题，上面的代码在调用 a.rp()的的时候，系统自动传递一了个 a 对象的指针给函数，在内部的时候 pp=100;的时候其实就是 this->pp=100;

所以你把上面的成员函数写成如下形势也是正确的：

```
void test::rp()
{
   ::pp=11;
   this->pp=100;//this 指针就是指向 a 对象的指针
}
```

类的成员函数和普通函数一样是可以进行重载操作的，关于重载函数前面已经说过，这里不再说明。

给出例子仔细看：

```
#include <iostream>
using namespace std;
class test
{
   private:
      int number;
   public:
      float socre;
      int pp;
   public:
      void rp(int);
      void rp(float);
};
void test::rp(int a)//在外部利用域区分符定义 test 类的成员函数
{
   cout<<"调用成员函数!a:"<<a<<endl;
}
void test::rp(float a)//在外部利用域区分符定义 test 类的成员函数
{
   cout<<"调用成员函数!a:"<<a<<endl;
}
```

```
void main()
{
   test a;
   a.rp(100);
   a.rp(3.14f);
   cin.get();
}
```

下面我们来看一下利用指针和利用引用间接调用类的成员函数，对于对于指针和引用调用成员函数和调用普通函数差别不大，在这里也就不再重复说明了，注意看代码，多试多练习既可。

代码如下：

```
#include <iostream>
using namespace std;
class test
{
   private:
      int number;
   public:
      float socre;
      int pp;
   public:
      int rp(int);
};
int test::rp(int a)//在外部利用域区分符定义 test 类的成员函数
{
   number=100;
   return a + number;
}

void run(test *p)//利用指针调用
{
   cout<<p->rp(100)<<endl;
}
void run(test &p)//利用引用
{
   cout<<p.rp(200)<<endl;
}

void main()
{
   test a;
   run(&a);
```

```
    run(a);
    cin.get();
}
```

前面我们说过，类的成员如果不显式的生命为 public 那么它默认的就是 private 就是私有的，私有声明可以保护成员不能够被外部访问，但在 c++还有一个修饰符，它具有和 private 相似的性能，它就是 protected 修饰符。

在这里我们简单说明一下，他们三着之间的差别：

在类的 private:节中声明的成员（无论数据成员或是成员函数）仅仅能被类的成员函数和友元访问。
在类的 protected: 节中声明的成员（无论数据成员或是成员函数）仅仅能被类的成员函数，友元以及子类的成员函数和友元访问。
在类的 public:节中声明的成员（无论数据成员或是成员函数）能被任何人访问。

由于 private 和 protected 的差别主要是体现在类的继承中，现在的教程还没有设计到友元和子类所以这里不做深入讨论，但上面的三点务必记得，在以后的教程中我们会回过头来说明的。

总的来说，类成员的保护无非是为了以下**四点**!

1. 相对与普通函数和其它类的成员函数来说，保护类的数据不能够被肆意的篡改侵犯!

2. 使类对它本身的内部数据维护负责，只有类自己才能够访问自己的保护数据!

3. 限制类的外部接口，把一个类分成公有的和受保护的两部分，对于使用者来说它只要会用就可以，无须了解内部完整结构，起到黑盒的效果。

4. 减少类与其它代码的关联程，类的功能是独立的，不需要依靠应用程序的运行环境，这个程序可以用它，另外一个也可以用它，使得你可以轻易的用一个类替换另一个类。

下面为了演示类成员的保护特性，我们来做一个球类游戏!

我们设计一个类，来计算球员的平均成绩，要求在外部不能够随意篡改球员的平均成绩。

我们把该类命名为 ballscore 并且把它放到 ballscore.h 的有文件中!

```
class ballscore
{
  protected:
    const static int gbs = 5;  //好球单位得分
    const static int bbs = -3;  //坏球单位扣分
    float gradescore;  //平均成绩
  public:
    float GetGS(float goodball,float badball)  //goodball 为好球数量,badball 为坏求数量
    {
```

```
        gradescore = (goodball*gbs + badball*bbs) / (goodball + badball);
        return gradescore;   //返回平均成绩
    }
};
```

主函数调用：

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

void main()
{
   ballscore jeff;
   cout<<jeff.GetGS(10,3);
   jeff.gradescore=5.5//想篡改 jeff 的平均成绩是错误的!
   cin.get();
}
```

在上面的代码中头文件和类的使用很好了体现了类的黑盒特性，谁也不能够在外部修改球员的平均成绩！

类体中的有一个地方要注意

```
const static int gbs = 5;//好球单位得分
const static int bbs = -3;//坏球单位扣分
```

之所以要修饰成 const static 因为 c++中类成员只有静态整形的常量才能够被初始化，到这里整个程序也就说完了，当然真正大比赛不可能是这样，只是为了说明问题就题命题而已，呵呵！

下面我们说一下关于类的作用域。

在说内容之前我们先给出这部分内容的一个完整代码，看讲解的是参照此一下代码。

```
#include <iostream>
using namespace std;
class ballscore
{
   protected:
      const static int gbs = 5;//好球单位得分
      const static int bbs = -3;//坏球单位扣分
      float gradescore;//平均成绩
   public:
      float GetGS(float goodball,float badball)   //goodball 为好球数量,badball 为坏求数量
```

```
    {
        int gradescore=0;
        //新定义一个和成员变量 float gradescore 相同名字的类成员函数局部变量
        ballscore::gradescore = (goodball*gbs + badball*bbs) /
         (goodball + badball);  //由于局部变量与类成员变量同名使用的时候必须在其前加上类名和域区分符
        return ballscore::gradescore;//返回平均成绩
    }
};
int ballscore=0;//定义一个与类名称相同的普通全局变量
int test;
void main()
{
  class test//局部类的创建
  {
    float a;
    float b;
  };
  test test;
  ::test=1;  //由于局部类名隐藏了外部变量使用需加域区分符
  class ballscore jeff;  //由于全局变量 int ballsocre 和类(ballsocre)名称相同,隐藏了类名称,这时候定义类对象需加 class 前缀以区分
  cout<<jeff.GetGS(10,3);
  cin.get();
}
```

类的作用域是只指定义和相应的成员函数定义的范围，在该范围内，一个类的成员函数对同一类的数据成员具有无限制的访问权。

在类的使用中，我们经常会碰到以下三种情况：

1.类的成员函数的局部变量隐藏了类的同名成员变量，看如对上面程序的分析。

```
protected:
    const static int gbs = 5;
    const static int bbs = -3;
    float gradescore;
public:
    float GetGS(float goodball,float badball)
    {
        int gradescore=0;
        ballscore::gradescore = (goodball*gbs + badball*bbs) /
        (goodball + badball);
        return ballscore::gradescore;
    }
```

代码中的 int gradescore 就把 float gradescore 给隐藏了，所以要使用成员变量 float gradescore 的时候必须在其之前加上类名称和域区分符(::)。

2. 在类定义外部非类型名隐藏了类型名称的情况，看上面代码的分析！

```
class ballscore
{
   protected:
      const static int gbs = 5;
      const static int bbs = -3;
      float gradescore;
   public:
      float GetGS(float goodball,float badball)
      {
         int gradescore=0;
         ballscore::gradescore = (goodball*gbs + badball*bbs) /
          (goodball + badball);
         return ballscore::gradescore;
      }
};
int ballscore=0;
```

代码中的全部变量 int ballscore 隐藏了类名称 class ballscore

所以在 main 中如如果要定义 ballscore 类的对象就要在类名称前加上 class 关键字

```
class ballscore jeff;
```

3. 类型名称隐藏了非类型名称，看对上面代码的分析

```
int test;
void main()
{
   class test
   {
      float a;
      float b;
   };
   test test;
   ::test=1;
   class ballscore jeff;
   cout<<jeff.GetGS(10,3);
   cin.get();
```

```
}
```

在普通函数内部定义的类叫做局部类，代码中的 test 类就是一个局部类！

代码中的 test 类隐藏了全局变量 test 如果要操作全局变量 test 那么就要在 test 前加上域区分符号(::)，进行使用！

::test=1 就是对全局变量 test 进行了赋值操作。

我们最后说一下名字空间！

名字空间就是指某一个名字在其中必须是唯一的作用域.

如果这个定义想不明白,可以简单的说成,在一个区域内,某一个名字在里面使用必须是唯一的,不能出现重复定义的情况出现,这个区域就是名字空间！

c++规定:

1.一个名字不能同时设置为两种不同的类型

```
class test
{
//...
};
typedef int test;
```

这个就是错误的！

2. 非类型名(变量名,常量名,函数名,对象名,枚举成员)不能重名.

```
test a;
void a();
```

就是错误的,因为 a 是一个 test 类的对象,它和函数 a 名称重名了！

3. 类型与非类型不在同一个名字空间上,可以重名,即使在同一作用域内,但两者同时出现时定义类对象的时候要加上前缀 class 以区分类型和非类型名！

```
class test
{
//.....
}

int test
```

```
class test a;//利用 class 前坠区分,定义了一个 test 类的对象 a
```

好了, 到这里关于类的知识点我们已经学习完, 希望大家多多练习

19.C++面向对象编程入门：构造函数与析构函数

请注意，这一节内容是 c++的重点，要特别注意！

我们先说一下什么是构造函数。

上一个教程我们简单说了关于类的一些基本内容，对于类对象成员的初始化我们始终是建立成员函数然后手工调用该函数对成员进行赋值的，那么在 c++中对于类来说有没有更方便的方式能够在对象创建的时候就自动初始化成员变量呢，这一点对操作保护成员是至关重要的，答案是肯定的。关于 c++类成员的初始化，有专门的构造函数来进行自动操作而无需要手工调用，在正式讲解之前先看看 c++对构造函数的一个基本定义。

1. C++规定，每个类必须有默认的构造函数，没有构造函数就不能创建对象。

2. 若没有提供任何构造函数，那么 c++提供自动提供一个默认的构造函数，该默认构造函数是一个没有参数的构造函数，它仅仅负责创建对象而不做任何赋值操作。

3. 只要类中提供了任意一个构造函数，那么 c++就不在自动提供默认构造函数。

4. 类对象的定义和变量的定义类似，使用默认构造函数创建对象的时候，如果创建的是静态或者是全局对象，则对象的位模式全部为 0，否则将会是随即的。

我们来看下面的代码：

```
#include <iostream>
using namespace std;
class Student
{
  public:
  Student()//无参数构造函数
  {
    number = 1;
    score = 100;
  }
  void show();

  protected:
  int number;
  int score;

};
```

```
void Student::show()
{
   cout<<number<<endl<<score<<endl;
}

void main()
{
   Student a;
   a.show();
   cin.get();
}
```

在类中的定义的和类名相同，并且没有任何返回类型的 Student()就是构造函数，这是一个无参数的构造函数，他在对象创建的时候自动调用，如果去掉 Student()函数体内的代码那么它和 c++的默认提供的构造函数等价的。

构造函数可以带任意多个的形式参数，这一点和普通函数的特性是一样的！

下面我们来看一个带参数的构造函数是如何进行对象的始化操作的。

代码如下：

```
#include <iostream>
using namespace std;
class Teacher
{
   public:
   Teacher(char *input_name)//有参数的构造函数
   {
      name=new char[10];
      //name=input_name;//这样赋值是错误的
      strcpy(name,input_name);
   }
   void show();

   protected:
   char *name;

};

void Teacher::show()
{
```

```
    cout<<name<<endl;
}

void main()
{
      //Teacher a;//这里是错误的,因为没有无参数的构造函数
    Teacher a("test");
    a.show();
    cin.get();
}
```

我们创建了一个带有字符指针的带有形参的 Teacher(char *input_name)的构造函数，调用它创建对象的使用类名加对象名称加扩号和扩号内参数的方式调用，这和调用函数有点类似，但意义也有所不同，因为构造函数是为创建对象而设立的，这里的意义不单纯是调用函数，而是创建一个类对象。

一旦类中有了一个带参数的构造函数而又没无参数构造函数的时候系统将无法创建不带参数的对象，所以上面的代码

```
Teacher a;
```

就是错误的！！！

这里还有一处也要注意：

```
//name=input_name;//这样赋值是错误的
```

因为 name 指是指向内存堆区的，如果使用 name=input_name;会造成指针指向改变不是指向堆区而是指向栈区，导致在后面调用析构函数 delete 释放堆空间出错！(析构函数的内容我们后面将要介绍)

如果需要调用能够执行就需要再添加一个没有参数的构造函数

对上面的代码改造如下：

```
#include <iostream>
using namespace std;
class Teacher
{
  public:
  Teacher(char *input_name)
  {
    name=new char[10];
    //name=input_name;//这样赋值是错误的
    strcpy(name,input_name);
  }
  Teacher()//无参数构造函数,进行函数重载
```

```
    {

    }
    void show();

    protected:
    char *name;

};

void Teacher::show()
{
    cout<<name<<endl;
}

void main()
{
    Teacher test;
    Teacher a("test");
    a.show();
    cin.get();
}
```

创建一个无阐述的同名的 Teacher()无参数函数，一重载方式区分调用，由于构造函数和普通函数一样具有重载特性所以编写程序的人可以给一个类添加任意多个构造函数，来使用不同的参数来进行初始话对象。

现在我们来说一下，一个类对象是另外一类的数据成员的情况，如果有点觉得饶人那么可以简单理解成:类成员的定义可以相互嵌套定义，一个类的成员可以用另一个类进行定义声明。

c++规定如果一个类对象是另外一类的数据成员，那么在创建对象的时候系统将自动调用那个类的构造函数。

下面我们看一个例子。

代码如下：

```
#include <iostream>
using namespace std;
class Teacher
{
    public:
    Teacher()
    {
        director = new char[10];
        strcpy(director,"王大力");
```

```
    }
    char *show();
    protected:
    char *director;
};
char *Teacher::show()
{
    return director;
}
class Student
{
    public:
    Student()
    {
        number = 1;
        score = 100;
    }
    void show();

    protected:
    int number;
    int score;
    Teacher teacher;//这个类的成员 teacher 是用 Teacher 类进行创建并初始化的

};

void Student::show()
{
    cout<<teacher.show()<<endl<<number<<endl<<score<<endl;
}

void main()
{
    Student a;
    a.show();
    Student b[5];
    for(int i=0; i<sizeof(b)/sizeof(Student); i++)
    {
        b[i].show();
    }
    cin.get();
}
```

上面代码中的 Student 类成员中 teacher 成员是的定义是用类 Teacher 进行定义创建的，那么系统碰到创建代码的时候就会自动调

用 Teacher 类中的 Teacher()构造函数对对象进行初始化工作！

这个例子说明类的分工很明确，只有碰到自己的对象的创建的时候才自己调用自己的构造函数。

一个类可能需要在构造函数内动态分配资源，那么这些动态开辟的资源就需要在对象不复存在之前被销毁掉，那么 c++类的析构函数就提供了这个方便。

析构函数的定义:析构函数也是特殊的类成员函数，它没有返回类型，没有参数，不能随意调用，也没有重载，只有在类对象的生命期结束的时候，由系统自动调用。

析构函数与构造函数最主要大不同就是在于调用期不同，构造函数可以有参数可以重载！

我们前面例子中的 Teacher 类中就使用 new 操作符进行了动态堆内存的开辟，由于上面的代码缺少析构函数，所以在程序结束后，动态开辟的内存空间并没有随着程序的结束而小时，如果没有析构函数在程序结束的时候逐一清除被占用的动态堆空间那么就会造成内存泄露，使系统内存不断减少系统效率将大大降低！

那么我们将如何编写类的析构函数呢？

析构函数可以的特性是在程序结束的时候逐一调用，那么正好与构造函数的情况是相反，属于互逆特性，所以定义析构函数因使用"~"符号(逻辑非运算符)，表示它为腻构造函数，加上类名称来定义。

看如下代码：

```
#include <iostream>
#include <string>
using namespace std;
class Teacher
{
  public:
  Teacher()
  {
    director = new char[10];
    strcpy(director,"王大力");
    //director = new string;
    // *director="王大力";//string 情况赋值
  }
  ~Teacher()
  {
    cout<<"释放堆区 director 内存空间 1 次";
    delete[] director;
    cin.get();
  }
  char *show();
```

```
    protected:
    char *director;
    //string *director;
};
char *Teacher::show()
{
    return director;
}
class Student
{
    public:
    Student()
    {
        number = 1;
        score = 100;
    }
    void show();

    protected:
    int number;
    int score;
    Teacher teacher;

};

void Student::show()
{
    cout<<teacher.show()<<endl<<number<<endl<<score<<endl;
}
void main()
{
    Student a;
    a.show();
    Student b[5];
    for(int i=0; i<sizeof(b)/sizeof(Student); i++)
    {
        b[i].show();
    }
    cin.get();
}
```

上面的代码中我们为 Teacher 类添加了一个名为~Teacher()的析构函数用于清空堆内存。

建议大家编译运行代码观察调用情况，程序将在结束前也就是对象生命周期结束的时候自动调用~Teacher()

~Teache()中的 delete[] director;就是清除堆内存的代码，这与我们前面一开始提到的。

name=input_name;//这样赋值是错误的

有直接的关系，因为 delete 操作符只能清空堆空间而不能清楚栈空间，如果强行清除栈空间内存的话将导致程序崩溃！

前面我们已经简单的说了类的构造函数和析构函数，我们知道一个类的成员可以是另外一个类的对象，构造函数允许带参数，那么我们可能会想到上面的程序我们可以在类中把 Student 类中的 teacher 成员用带参数的形式调用 Student 类的构造函数，不必要再在 Teacher 类中进行操作，由于这一点构想我们把程序修改成如下形式：

```
#include <iostream>
#include <string>
using namespace std;
class Teacher
{
  public:
  Teacher(char *temp)
  {
    director = new char[10];
    strcpy(director,temp);
  }
  ~Teacher()
  {
    cout<<"释放堆区 director 内存空间 1 次";
    delete[] director;
    cin.get();
  }
  char *show();
  protected:
  char *director;
};
char *Teacher::show()
{
  return director;
}
class Student
{
  public:
  Student()
  {
    number = 1;
    score = 100;
  }
```

```
    void show();

    protected:
    int number;
    int score;
    Teacher teacher("王大力");//错误，一个类的成员如果是另外一个类的对象的话，不能在类中使用带参数的构造函数进行初始化

};

void Student::show()
{
    cout<<teacher.show()<<endl<<number<<endl<<score<<endl;
}
void main()
{
    Student a;
    a.show();
    Student b[5];
    for(int i=0; i<sizeof(b)/sizeof(Student); i++)
    {
        b[i].show();
    }
    cin.get();
}
```

可是很遗憾，程序不能够被编译成功,为什么呢？

因为：类是一个抽象的概念，并不是一个实体，并不能包含属性值(这里来说也就是构造函数的参数了)，只有对象才占有一定的内存空间，含有明确的属性值！

这一个问题是类成员初始化比较尴尬的一个问题，是不是就没有办法解决了呢？呵呵。。。。。。

c++为了解决此问题，有一个很独特的方法，下一小节我们将介绍。

对于上面的那个"尴尬"问题，我们可以在构造函数头的后面加上:号并指定调用哪那个类成员的构造函数来解决！

教程写到这里的时候对比了很多书籍，发现几乎所有的书都把这一章节叫做**构造类成员**，笔者在此觉得有所不妥，因为从读音上容易混淆概念，所以把这一小节的名称改为**构造类的成员**比较合适！

代码如下：

```
#include <iostream>
```

```
using namespace std;
class Teacher
{
   public:
   Teacher(char *temp)
   {
      director = new char[10];
      strcpy(director,temp);
   }
   ~Teacher()
   {
      cout<<"释放堆区 director 内存空间 1 次";
      delete[] director;
      cin.get();
   }
   char *show();
   protected:
   char *director;
};
char *Teacher::show()
{
   return director;
}
class Student
{
   public:
   Student(char *temp):teacher(temp)
   {
      number = 1;
      score = 100;
   }
   void show();

   protected:
   int number;
   int score;
   Teacher teacher;

};

void Student::show()
{
   cout<<teacher.show()<<endl<<number<<endl<<score<<endl;
}
```

```
void main()
{
   Student a("王大力");
   a.show();
   //Student b[5]("王大力"); //这里这么用是不对的,数组不能够使用带参数的构造函数,以后我们将详细介绍 vector 类型
   // for(int i=0; i<sizeof(b)/sizeof(Student); i++)
   //{
   //    b[i].show();
   //}
   cin.get();
}
```

大家可以发现最明显的改变在这里

```
Student(char *temp):teacher(temp)
```

冒号后的 teacher 就是告诉调用 Student 类的构造函数的时候把参数传递给成员 teacher 的 Teacher 类的构造函数，这样一来我们就成功的在类体外对 teacher 成员进行了初始化，既方便也高效，这种冒号后指定调用某成员构造函数的方式，可以同时制定多个成员，这一特性使用逗号方式，例如：

```
Student(char *temp):teacher(temp),abc(temp),def(temp)
```

由冒号后可指定调用哪那个类成员的构造函数的特性，使得我们可以给类的常量和引用成员进行初始化成为可能。

我们修改上面的程序，得到如下代码：

```
#include <iostream>
#include <string>
using namespace std;
class Teacher
{
   public:
   Teacher(char *temp)
   {
      director = new char[10];
      strcpy(director,temp);
   }
   ~Teacher()
   {
      cout<<"释放堆区 director 内存空间 1 次";
      delete[] director;
      cin.get();
   }
   char *show();
```

```
    protected:
    char *director;
};
char *Teacher::show()
{
    return director;
}
class Student
{
    public:
    Student(char *temp,int &pk):teacher(temp),pk(pk),ps(10)
    {
        number = 1;
        score = 100;
    }
    void show();

    protected:
    int number;
    int score;
    Teacher teacher;
    int &pk;
    const int ps;

};

void Student::show()
{
    cout<<teacher.show()<<endl<<number<<endl<<score<<endl<<pk<<endl<<ps<<endl;
}
void main()
{
    char *t_name="王大力";
    int b=99;
    Student a(t_name,b);
    a.show();
    cin.get();
}
```

改变之处最重要的在这里 Student(char *temp,int &pk):teacher(temp),pk(pk),ps(10)

调用的时候我们使用

```
Student a(t_name,b);
```

我们将 b 的地址传递给了 int &pk 这个引用，使得 Student 类的引用成员 pk 和常量成员 ps 进行了成功的初始化。

但是细心的人会发现，我们在这里使用的初始化方式并不是在构造函数内进行的，而是在外部进行初始化的，的确，在冒号后和在构造函数括号内的效果是一样的，但和 teacher(temp)所不同的是，pk(pk)的括号不是调用函数的意思，而是赋值的意思，我想有些读者可能不清楚新标准的 c++对变量的初始化是允许使用括号方式的，int a=10 和 int a(10)的等价的，但冒号后是不允许使用=方式只允许()括号方式，所以这里只能使用 pk(pk)而不能是 pk=pk 了。

这一小节的内容是说**对象构造的顺序**的，对象构造的顺序直接关系程序的运行结果，有时候我们写的程序不错，但运行出来的结果却超乎我们的想象，了解 c++对对象的构造顺序有助于解决这些问题。

c++规定，所有的全局对象和全局变量一样都在主函数 main()之前被构造，函数体内的静态对象则只构造一次，也就是说只在首次进入这个函数的时候进行构造！

代码如下：

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

class Test
{
public:
   Test(int a)
   {
      kk=a;
      cout<<"构造参数 a:"<<a<<endl;
   }
public:
   int kk;
};

void fun_t(int n)
{
   static Test a(n);
   //static Test a=n;//这么写也是对的
   cout<<"函数传入参数 n:"<<n<<endl;
   cout<<"对象 a 的属性 kk 的值:"<<a.kk<<endl;
}
Test m(100);
void main()
{
```

```
    fun_t(20);
    fun_t(30);
    cin.get();
}
```

下面我们来看一下，**类成员的构造顺序**的问题。

先看下面的代码：

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

class Test
{
public:
    Test(int j):pb(j),pa(pb+5)
    {

    }
public:
    int pa;
    int pb;
};
void main()
{
    Test a(10);
    cout<<a.pa<<endl;
    cout<<a.pb<<endl;
    cin.get();
}
```

上面的程序在代码上是没有任何问题的，但运行结果可能并不如人意。

pa 并没有得到我们所希望的 15 而是一个随机的任意地址的值。

这又是为什么呢？

类成员的构造是按照在类中定义的顺序进行的，而不是按照构造函数说明后的冒号顺序进行构造的，这一点需要记住！

20.理解 C++面向对象程序设计中的抽象理论

很多书在一开始就开始学习 josephus 问题，为了让大家前面学起来较为容易我把前面涉及到此问题的地方都故意去掉了，现在我们已经学习过了结构体和类，所以放在这里学习可能更合适一些。

在正式开始学习之前我们先回顾一下如何利用数组和结构体的方式来解决，最后我们再看一下如何利用面向对象的抽象理念进行解决此问题的程序设计，相互对比，找出效率最高，最容易理解，最方便维护的程序来，说明利用面向对象的抽象理念进行程序设计的好处。

josephus 问题其实就是一个游戏，一群小孩围成一个圈，设置一个数，这个数是个小于小孩总数大于 0 的一个整数，从第一个小孩开始报数，当其中一个小孩报到你设置的那个数的时候离开那个圈，这样一来反复报下去，直到只剩下最后一个小孩的时候那个小孩就是胜利者，写程序来找出这个小孩。

以下是数组方法：

由于数组的限制我们必须预先假设好有多少个小孩，离开的小孩他自身设置为 0 来标记离开状态。

代码如下：

```
#include <iostream>
using namespace std;
void main()
{
 const int num=10;
 int interval;
 int a[num];
 for(int i=0; i<num; i++)
 {
  a[i]=i+1;
 }
  cout <<"please input the interval: ";
 cin >>interval;
 for(int i=0; i<num; i++)
 {
  cout <<a[i] <<",";
 }
  cout <<endl;

int k=1;
int p=-1;
```

```
while(1)
{
   for(int j=0;j<interval;)
   {
      p=(p+1)%num;
      if(a[p]!=0)
      {
         j++;
      }
   }
   if(k==num)
   {
      break;
   }
   cout<<a[p]<<",";
   a[p]=0;
   k++;
}
cout <<"\nNo." <<a[p] <<" boy've won.\n";
cin.get();
cin.get();
}
```

就数组解决来看,程序简短但效率不高可读性也不好，此代码没有什么特别之处主要依靠一个加 1 取模的方式来回到首位置，形成环链:p=(p+1)%num;。

以下是利用结构体的方法解决 josephus 问题：

当我们学过结构体后，我们了解到结构体自身的成员指针可以指向自身对象的地址的时候，我们很容易想到解决这个数学问题，用结构体来描述是再合适不过的了，用它可以很完美的描述环形链表。

代码如下：

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

struct Children
{
   int number;
   Children *next;
};

void show(Children *point,int num)//环链输出函数
```

```
{
    for(int i=1;i<=num;i++)
    {
        cout<<point->number<<",";
        point = point->next;
    }
}

void main()
{
    int num;//孩子总数
    int interval;//抽选号码
    cout<<"请输入孩子总数:";
    cin>>num;
    cout<<"请输入抽选号码:";
    cin>>interval;

    Children *josephus = new Children[num];//设置圈的起点指针,并动态开辟堆空间用于存储数据

    Children *point = josephus;//用于初化链表的指针,起始地址与 josephus 指针相同

    for(int i=1;i<=num;i++)
    {
        point -> number = i;
        point -> next = josephus + i % num;//利用+1 取模的方式设置节点的 next 指针,当到最后的时候自动指向到第一个,形成环链
        point = point->next;//将位置移到下一饿节点也就是下一个小孩的位置
    }

    show(point,num);

    Children *cut_point;
    point=&josephus[num-1];//把起始指针设置在最后一个节点,当进入循环的时候就会从 0 开始,这样就好让不需要的节点脱离
    int k=0;//故意设置一个 k 观察 while 循环了多少次
    while(point->next!=point)//通过循环不断的寻找需要放弃的节点
    {
        k++;
        for(int i = 0;i<interval;i++)//找需要放弃的节点位置
        {
            cut_point=point;//存储截断位置指针
            point=cut_point->next;//将 point 的指针移动到放弃的节点位置,此处也和 while 循环终止条件有关系
        }
        cut_point->next=point->next;//将截断出的 next 指针设置成放弃处节点的 next 指针,使放弃处节点也就是不需要的节点脱离
        cout<<"k:"<<k<<endl;
    }
```

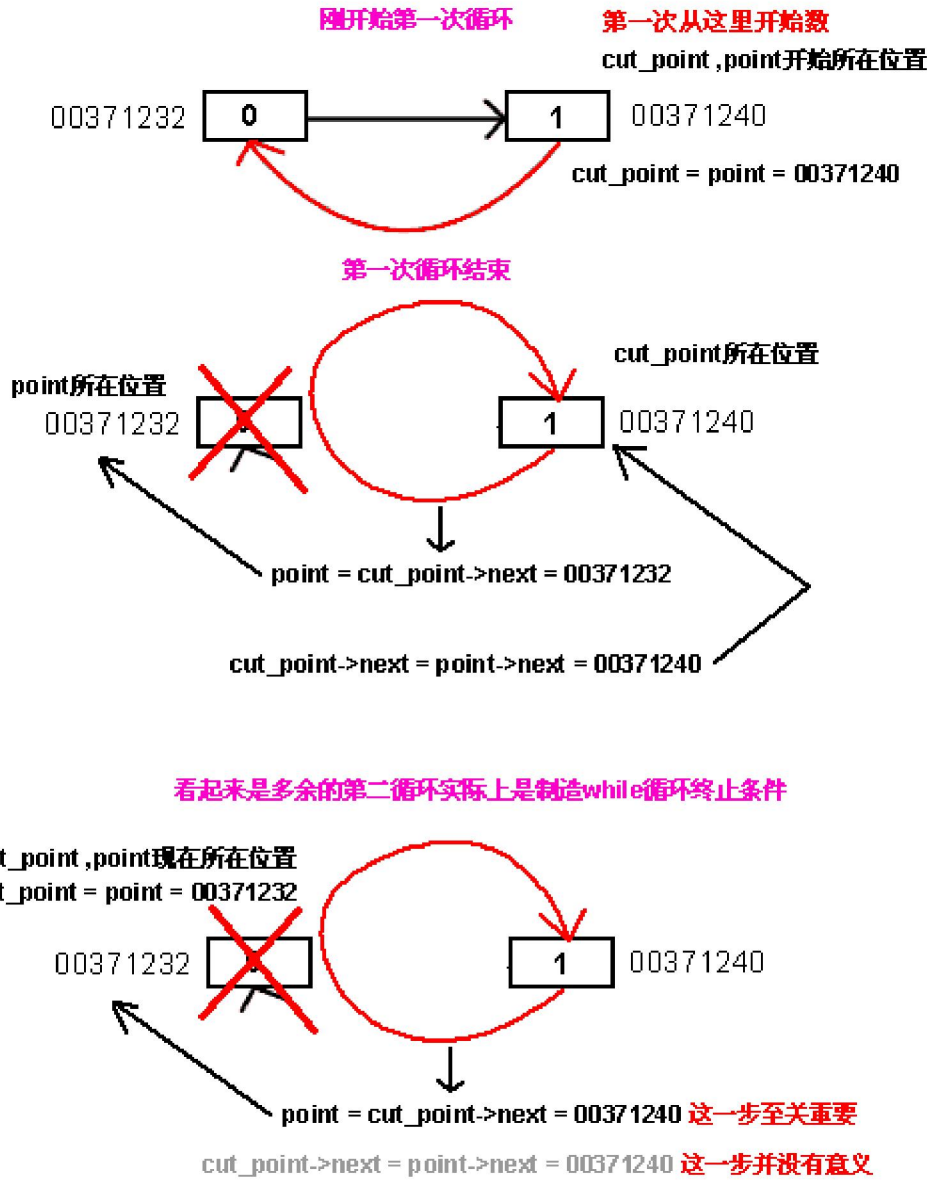
```
    cout<<"\n 最后的赢家:"<<endl<<point->number<<endl<<point<<endl<<point->next<<endl;
    delete[] josephus;
    cin.get();
    cin.get();
}
```

此代码较为难以理解的部分就是 while 循环的终止条件的设置，如果读者没有能够理解好这部分注意看下面的图式帮助学习。

结构体的解法非常重要，对于我们全面理解面向对象的程序设计的抽象问题是基础，必须看明白我们才能够进行后面知识的学习,务必认真对待。

这段代码比较前一个程序，可读性上有所加强，但仍然不太容易理解！

输入两个小孩每次都去掉第一个小孩的情况



这次看似多余的循环就是创造while循环终止条件使point指针回到位置1上,构成了point->next==point

为了更容易学习便于理解，我们的图例是以有两个小孩围成一圈，并且设置报数的数为 1 的情况来制作的。

上面的两种解决 Josephus 问题的解决办法从代码上来看，都属于一杆子到底的解法，第二种从结构表达上优于第一种，但是这两个都属于纯粹的过程式程序设计，程序虽然简短,但很难让人看懂，程序的可读性不高，在我们没有学习面向对象的编程之前，聪明的人可

能会把各各步骤分解出来做成由几个函数来解决问题。

思路大致可以分为以下六个部分：

1. 建立结构

2. 初始化小孩总数,和数小孩的数

3. 初始化链表并构成环链

4. 开始通过循环数小孩获得得胜者

5. 输出得胜者

6. 返回堆内存空间

从表上看这个程序为了便于阅读可以写成六个函数来分别处理这六个过程，的确，这么修改过后程序的可读性是提高了一大步，但是有缺点仍然存在，程序完全暴露在外，任何人都可以修改程序，程序中的一些程序作者不希望使用者能够修改的对象暴露在外，各对象得不到任何的保护，不能保证程序在运行中不被意外修改，对于使用者来说还是需要具备解决 Josephus 问题算法的能力，一旦程序变的越来越很，,每一个参与开发的程序员都需要通读程序的所有部分，程序完全不具备黑盒效应，给多人的协作开发带来了很大的麻烦，几乎每个人都做了同样的重复劳动，这种为了解决一个分枝小问题写一个函数，最后**由很多个解决局部问题的函数**组合成的程序我们叫做**结构化程序设计**，结构化编程较过程化编程相比可读性是提高了，但程序不能轻易的被分割解决一个个大问题的模块，在主函数中使用他们的时候总是这个函数调用到那个函数，如果你并不是这些函数的作者就很难正确方便的使用这些函数，而且程序的变量重名问题带来的困扰也是很让人头痛的……

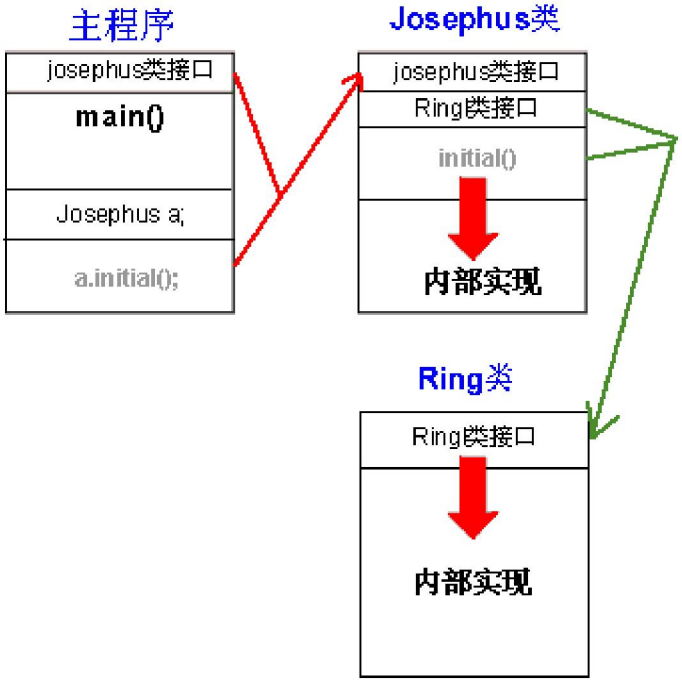
那么面向对象的程序设计又是如何解决这些问题的呢？

面向对象的程序设计的思路是这样的：

程序 = 对象 + 对象 +对象.....

这么组合而来的

对于上面的 josephus 问题可以把问题分割成如下的方法进行设计(如下图所示)



附件： 点击下载(11K, zip 压缩文件)

由上图可以看出：

面向对象的程序设计是由类组合而成的，有类必然有类的对象，程序之间的交互主要是通过对象与对象之间的关系进行操作的。

由于我们把 josephus 问题分解成了 josephus 类和 ring 类，在主函数中，用户只需要使用 josephus 类设计其对象明确知道 Josephus 类的外部接口函数也就是操作该对象的方法 initial()就可以了，至于 josephus 的内部实现又是如何与 Ring 类进行操作的使用者一概不需要知道，只要拿来用知道接口和接口函数是什么就可以了，这样的程序设计很好的保护了各类成员数据的安全，主函数代码调用极其简单只有建立对象和调用对象方法的操作这两部而已，以后类一旦需要修改，只修改类体本身就可以，而主函数不需要做任何修改，这样就很好的做到了什么人做的事情什么人处理互不冲突。

程序的代码如下，我把工程文件压缩了作为此帖的附件提供下载，希望读者仔细阅读仔细推敲，真正理解面向对象 oop 编程的特点和意图。

主程序 test4.cpp

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

void main()
{
   Josephus a;
```

```
    a.initial();
    cin.get();
    cin.get();
}
```

josephus.h

```
class Josephus
{
public:
    Josephus(int num=10,int interval=1)
    {
        Josephus::num=num;
        Josephus::interval=interval;
    }
    void initial();
protected:
    int num;
    int interval;
};
```

josephus.cpp

```
#include <iostream>
#include "josephus.h"
#include "ring.h"

using namespace std;

void Josephus::initial()
{
    int num,interval;
    cout<<"请输入孩子总数:";
    cin>>num;
    if(num<2)
    {
        cout<<"孩子总数不能小于 2,否则不能构成环链!";
        return;
    }
    cout<<"请输入抽选号码";
    cin>>interval;
    if(interval<1|interval>num)
    {
```

```
        cout<<"请输入抽选号码不能小于 1 或者大于小孩总数!";
        return;
    }
    Josephus::num=num;
    Josephus::interval=interval;
    Ring a(num);
    a.ShowRing(num);
    cout<<endl;
    for(int i=1;i<num;i++)
    {
        a.CountInterval(interval);
        a.ShowWiner_loser();
        a.OutChild();
    }
    cout<<endl<<"胜利者是:";
    a.ShowWiner_loser();
}
```

ring.h

```
struct Children
{
    int number;
    Children *next;
};

class Ring
{
public:
    Ring(int num)
    {
        josephus = new Children[num];
        point = josephus;
        for(int i=1;i<=num;i++)
        {
            point->number = i;
            point->next = josephus + i % num;
            point=point->next;
        }
        point = &josephus[num-1];
    }
    ~Ring()
    {
        delete[] josephus;
```

```
    }
    void ShowRing(int num);
    void CountInterval(int interval);
    void OutChild();
    void ShowWiner_loser();
protected:
    Children *josephus;
    Children *point;
    Children *cut_point;
};
```

ring.cpp

```
#include <iostream>
#include "ring.h"

using namespace std;
void Ring::ShowRing(int num)
{
        point=josephus;//也可以写成 point=point->next;但前着效率高一点点
        for(int i=1;i<=num;i++)
        {
            cout<<point->number<<",";
            point=point->next;
        }
        point=&josephus[num-1];//输出过后恢复 point 应该在的位置
}
void Ring::CountInterval(int interval)//数小孩
{
    for(int i=0;i<interval;i++)
    {
        cut_point = point;
        point = cut_point->next;
    }
}
void Ring::OutChild()
{
    cut_point->next = point->next;//将不要节点断离
    point=cut_point;
}
void Ring::ShowWiner_loser()
{
    cout<<point->number<<",";
```

```
}
```

程序中需要注意的小地方是在这里

```
class Josephus
{
public:
    Josephus(int num=10,int interval=1)
    {
       Josephus::num=num;
       Josephus::interval=interval;
    }
    void initial();
protected:
   int num;
   int interval;
};
```

代码中的

```
Josephus::num=num;
Josephus::interval=interval;
```

使用域区分符的目的就是为了区分成员变量和局部变量 Josephus(int num=10, int interval=1)

相信读者认真读完程序认真理解后应该就可以理解面向对象程序设计的用意和好处了，切记认真推敲！

大家看到面向对象程序设计的解决办法，可能觉得它的代码太多了，会怀疑它执行的效率是否足够好，呵呵！

这里只能这么说，程序的效率不是单单看程序的长短来看的，优秀的程序应该是便于维护，关系清楚的，面向对象的程序设计其实和过程式或者是结构化程序设计的思路是不冲突的，在不同的地方使用不同的方法，优势互补才是正道！

21.C++类对象的复制－拷贝构造函数

在学习这一章内容前我们已经学习过了类的构造函数和析构函数的相关知识，对于普通类型的对象来说，他们之间的复制是很简单的，例如：

```
int a = 10;
int b =a;
```

自己定义的类的对象同样是对象，谁也不能阻止我们用以下的方式进行复制, 例如：

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

class Test
{
public:
   Test(int temp)
   {
      p1=temp;
   }
protected:
   int p1;

};

void main()
{
   Test a(99);
   Test b=a;
}
```

普通对象和类对象同为对象，他们之间的特性有相似之处也有不同之处，类对象内部存在成员变量，而普通对象是没有的，当同样的复制方法发生在不同的对象上的时候，那么系统对他们进行的操作也是不一样的，就类对象而言，相同类型的类对象是通过**拷贝构造函数**来完成整个复制过程的，在上面的代码中, 我们并没有看到拷贝构造函数，同样完成了复制工作，这又是为什么呢？因为当一个类没有自定义的拷贝构造函数的时候系统会自动提供一个**默认的拷贝构造函数**，来完成复制工作。

下面，我们为了说明情况，就普通情况而言(以上面的代码为例)，我们来自己定义一个与系统默认拷贝构造函数一样的拷贝构造函数，看看它的内部是如何工作的！

代码如下：

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

class Test
{
public:
   Test(int temp)
   {
      p1=temp;
   }
   Test(Test &c_t)//这里就是自定义的拷贝构造函数
   {
      cout<<"进入 copy 构造函数"<<endl;
      p1=c_t.p1;//这句如果去掉就不能完成复制工作了,此句复制过程的核心语句
   }
public:
   int p1;
};

void main()
{
   Test a(99);
   Test b=a;
   cout<<b.p1;
   cin.get();
}
```

上面代码中的 Test(Test &c_t)就是我们自定义的拷贝构造函数，拷贝构造函数的名称必须与类名称一致，函数的形式参数是本类型的一个引用变量,**且必须是引用**。

当用一个已经初始化过了的自定义类类型对象去初始化另一个新构造的对象的时候，拷贝构造函数就会被自动调用，如果你没有自定义拷贝构造函数的时候系统将会提供给一个默认的拷贝构造函数来完成这个过程，上面代码的复制核心语句就是通过 Test(Test &c_t)拷贝构造函数内的 p1=c_t.p1;语句完成的。如果取掉这句代码，那么 b 对象的 p1 属性将得到一个未知的随机值；

下面我们来讨论一下关于浅拷贝和深拷贝的问题。

就上面的代码情况而言，很多人会问到，既然系统会自动提供一个默认的拷贝构造函数来处理复制，那么我们没有意义要去自定义拷贝构造函数呀，对，就普通情况而言这的确是没有必要的，但在某写状况下，类体内的成员是需要开辟动态开辟堆内存的,如果我们不自定义拷贝构造函数而让系统自己处理，那么就会导致堆内存的所属权产生混乱，试想一下，已经开辟的一端堆地址原来是属于对象 a 的，由于复制过程发生，b 对象取得是 a 已经开辟的堆地址，一旦程序产生析构，释放堆的时候，计算机是不可能清楚这段地址是真正属于谁的，当连续发生两次析构的时候就出现了运行错误。

为了更详细的说明问题，请看如下的代码。

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

class Internet
{
public:
   Internet(char *name,char *address)
   {
      cout<<"载入构造函数"<<endl;
      strcpy(Internet::name,name);
      strcpy(Internet::address,address);
      cname=new char[strlen(name)+1];
      if(cname!=NULL)
      {
         strcpy(Internet::cname,name);
      }
   }
   Internet(Internet &temp)
   {
      cout<<"载入 COPY 构造函数"<<endl;
      strcpy(Internet::name,temp.name);
      strcpy(Internet::address,temp.address);
      cname=new char[strlen(name)+1];//这里注意,深拷贝的体现!
      if(cname!=NULL)
      {
         strcpy(Internet::cname,name);
      }
   }
   ~Internet()
   {
      cout<<"载入析构函数!";
      delete[] cname;
      cin.get();
   }
   void show();
protected:
   char name[20];
   char address[30];
   char *cname;
};
void Internet::show()
{
```

```
    cout<<name<<":"<<address<<cname<<endl;
}
void test(Internet ts)
{
    cout<<"载入 test 函数"<<endl;
}
void main()
{
    Internet a("中国软件开发实验室","www.cndev-lab.com");
    Internet b = a;
    b.show();
    test(b);
}
```

上面代码就演示了深拷贝的问题,对对象 b 的 cname 属性采取了新开辟内存的方式避免了内存归属不清所导致析构释放空间时候的错误,最后我必须提一下，对于上面的程序我的解释并不多，就是希望读者本身运行程序观察变化，进而深刻理解。

拷贝和浅拷贝的定义可以简单理解成：如果一个类拥有资源(堆，或者是其它系统资源)，当这个类的对象发生复制过程的时候，这个过程就可以叫做**深拷贝**，反之对象存在资源但复制过程并未复制资源的情况视为**浅拷贝**。

浅拷贝资源后在释放资源的时候会产生资源归属不清的情况导致程序运行出错,这点尤其需要注意!

以前我们的教程中讨论过函数返回对象产生临时变量的问题，接下来我们来看一下**在函数中返回自定义类型对象**是否也遵循此规则产生临时对象!

先运行下列代码：

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

class Internet
{
public:
    Internet()
    {

    };
    Internet(char *name,char *address)
    {
        cout<<"载入构造函数"<<endl;
        strcpy(Internet::name,name);
    }
    Internet(Internet &temp)
```

```
    {
        cout<<"载入 COPY 构造函数"<<endl;
        strcpy(Internet::name,temp.name);
        cin.get();
    }
    ~Internet()
    {
        cout<<"载入析构函数!";
        cin.get();
    }
protected:
    char name[20];
    char address[20];
};
Internet tp()
{
    Internet b("中国软件开发实验室","www.cndev-lab.com");
    return b;
}
void main()
{
    Internet a;
    a=tp();
}
```

从上面的代码运行结果可以看出，程序一共载入过析构函数三次，证明了由函数返回自定义类型对象同样会产生临时变量，事实上对象 a 得到的就是这个临时 Internet 类类型对象 temp 的值。

这一下节的内容我们来说一下**无名对象**。

利用无名对象初始化对象系统不会不调用拷贝构造函数。

那么什么又是无名对象呢？

很简单，如果在上面程序的 main 函数中有：

Internet ("中国软件开发实验室","www.cndev-lab.com");

这样的一句语句就会产生一个无名对象，无名对象会调用构造函数但利用无名对象初始化对象系统不会不调用拷贝构造函数！

下面三段代码是很见到的三种利用无名对象初始化对象的例子。

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

```
class Internet
{
public:
   Internet(char *name,char *address)
   {
      cout<<"载入构造函数"<<endl;
      strcpy(Internet::name,name);
   }
   Internet(Internet &temp)
   {
      cout<<"载入 COPY 构造函数"<<endl;
      strcpy(Internet::name,temp.name);
      cin.get();
   }
   ~Internet()
   {
      cout<<"载入析构函数!";
   }
public:
   char name[20];
   char address[20];
};

void main()
{
   Internet a=Internet("中国软件开发实验室","www.cndev-lab.com");
   cout<<a.name;
   cin.get();
}
```

上面代码的运行结果有点“出人意料”，从思维逻辑上说，当无名对象创建了后，是应该调用自定义拷贝构造函数，或者是默认拷贝构造函数来完成复制过程的，但事实上系统并没有这么做，因为无名对象使用过后在整个程序中就失去了作用，对于这种情况 c++会把代码看成是：

```
Internet a("中国软件开发实验室",www.cndev-lab.com);
```

省略了创建无名对象这一过程，所以说不会调用拷贝构造函数。

最后让我们来看看引用无名对象的情况。

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

```
class Internet
{
public:
   Internet(char *name,char *address)
   {
      cout<<"载入构造函数"<<endl;
      strcpy(Internet::name,name);
   }
   Internet(Internet &temp)
   {
      cout<<"载入 COPY 构造函数"<<endl;
      strcpy(Internet::name,temp.name);
      cin.get();
   }
   ~Internet()
   {
      cout<<"载入析构函数!";
   }
public:
   char name[20];
   char address[20];
};

void main()
{
   Internet &a=Internet("中国软件开发实验室","www.cndev-lab.com");
   cout<<a.name;
   cin.get();
}
```

引用本身是对象的别名，和复制并没有关系，所以不会调用拷贝构造函数,但要注意的是，在 c++看来:

```
Internet &a=Internet("中国软件开发实验室","www.cndev-lab.com");
```

是等价与:

```
Internet a("中国软件开发实验室","www.cndev-lab.com");
```

的,注意观察调用析构函数的位置（这种情况是在 main()外调用，而无名对象本身是在 main()内析构的）。

22.C++类静态数据成员与类静态成员函数

在没有讲述本章内容之前如果我们想要在一个范围内共享某一个数据，那么我们会设立全局对象，但面向对象的程序是由对象构成的，我们如何才能在类范围内共享数据呢？

这个问题便是本章的重点：

声明为 static 的类成员或者成员函数便能在类的范围内共同享，我们把这样的成员称做**静态成员和静态成员函数**。

下面我们用几个实例来说明这个问题，类的成员需要保护，通常情况下为了不违背类的封装特性，我们是把类成员设置为 protected（保护状态）的，但是我们为了简化代码，使要说明的问题更为直观，更容易理解，我们在此处都设置为 public。

以下程序我们来做一个模拟访问的例子,在程序中，每建立一个对象我们设置的类静态成员变自动加一，代码如下：

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

class Internet
{
public:
   Internet(char *name,char *address)
   {
      strcpy(Internet::name,name);
      strcpy(Internet::address,address);
      count++;
   }
   static void Internet::Sc()//静态成员函数
   {
      cout<<count<<endl;
   }
   Internet &Rq();
public:
   char name[20];
   char address[20];
   static int count;//这里如果写成 static int count=0;就是错误的
};

Internet& Internet::Rq()//返回引用的成员函数
{
   return *this;
```

```
}

int Internet::count = 0;//静态成员的初始化
void vist()
{
   Internet a1("中国软件开发实验室","www.cndev-lab.com");
   Internet a2("中国软件开发实验室","www.cndev-lab.com");
}
void fn(Internet &s)
{
   cout<<s.Rq().count;
}
void main()
{
   cout<<Internet::count<<endl;//静态成员值的输出
   vist();
   Internet::Sc();//静态成员函数的调用
   Internet b("中国软件开发实验室","www.cndev-lab.com");
   Internet::Sc();
   fn(b);
   cin.get();
}
```

上面代码我们用了几种常用的方式建立对象，当建立新对象并调用其构造函数的时候，静态成员 cout 便运行加 1 操作,静态成员的初始化应该在主函数调用之前，并且不能在类的声明中出现，通过运行过程的观察我们发现，静态成员 count 的状态并不会随着一个新的对象的新建而重新定义，尽而我们了解到类的静态成员是属于类的而不是属于哪一个对象的，所以静态成员的使用应该是**类名称加域区分符加成员名称**的，在上面的代码中就是 Internet::count，虽然我们仍然可以使用对象名加点操作符号加成员名称的方式使用，但是不推荐的，静态态类成员的特性就是属于类而不专属于某一个对象。

静态成员函数的特性类似于静态成员的使用,同样与对象无关，调用方法为**类名称加域区分符加成员函数名称**，在上面的代码中就是 Internet::Sc();，静态成员函数由于与对象无关系，所以在其中是不能对类的普通成员进行直接操作的。

如果上面的 static void Internet::Sc()修改成为：

```
static void Internet::Sc()//静态成员函数
{
   cout<<name<<endl;//错误
   cout<<count<<endl;
}
```

静态成员函数与普通成员函数的差别就在于缺少 this 指针，没有这个 this 指针自然也就无从知道 name 是哪一个对象的成员了。

根据类静态成员的特性我们可以简单归纳出几点，静态成员的使用范围：

1. 用来保存对象的个数。

2. 作为一个标记，标记一些动作是否发生，比如：文件的打开状态，打印机的使用状态，等等。

3. 存储链表的第一个或者最后一个成员的内存地址。

为了做一些必要的练习，深入的掌握静态对象的存在的意义，我们以前面的结构体的教程为基础，用类的方式描述一个线性链表，用于存储若干学生的姓名，代码如下：

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

class Student
{
public:
   Student (char *name);
   ~Student();
public:
   char name[30];
   Student *next;
   static Student *point;
};

Student::Student (char *name)
{
   strcpy(Student::name,name);
   this->next=point;
   point=this;
}

Student::~Student ()//析构过程就是节点的脱离过程
{
   cout<<"析构:"<<name<<endl;

   if(point==this)
   {
      point=this->next;
      cin.get();
      return;
   }
   for(Student *ps=point;ps;ps=ps->next)
   {
      if(ps->next==this)
      {
```

```
        cout<<ps->next<<"|"<<this->next<<endl;
        ps->next=next;//=next 也可以写成 this->next;
        cin.get();
        return;
        }
    }
    cin.get();
}

Student* Student::point=NULL;
void main()
{
    Student *c = new Student("marry");
    Student a("colin");
    Student b("jamesji");
    delete c;
    Student *fp=Student::point;
    while(fp!=NULL)
    {
        cout<<fp->name<<endl;
        fp=fp->next;
    }
    cin.get();
}
```

从上面的代码来看，原来单纯结构化编程需要的一个链表进入全局指针在这里被类的静态成员指针所替代(类的静态成员完全可以替代全局变量)，这个例子的理解重点主要是要注意观察类成员的析构顺序，通过对**析构顺序**的理解，使用析构函数来进行节点的脱链操作。

23.入门教程：实例详解 C++友元

在说明什么是友元之前，我们先说明一下为什么需要友元与友元的缺点：

通常对于普通函数来说,要访问类的保护成员是不可能的，如果想这么做那么必须把类的成员都生命成为 public(共用的)，然而这做带来的问题遍是任何外部函数都可以毫无约束的访问它操作它，c++利用 friend 修饰符，可以让一些你设定的函数能够对这些保护数据进行操作，避免把类成员全部设置成 public，最大限度的保护数据成员的安全。

友元能够使得普通函数直接访问类的保护数据，避免了类成员函数的频繁调用，可以节约处理器开销，提高程序的效率，但矛盾的是，即使是最大限度的保护，同样也破坏了类的封装特性，这即是**友元的缺点**，在现在 cpu 速度越来越快的今天我们并不推荐使用它，但它作为 c++一个必要的知识点，一个完整的组成部分，我们还是需要讨论一下的。

在类里声明一个普通函数，在前面加上 friend 修饰，那么这个函数就成了该类的友元，可以访问该类的一切成员。

下面我们来看一段代码，看看我们是如何利用友元来访问类的一切成员的。

```
#include <iostream>
using namespace std;
class Internet
{
public:
   Internet(char *name,char *address)
   {
      strcpy(Internet::name,name);
      strcpy(Internet::address,address);
   }
friend void ShowN(Internet &obj);//友元函数的声明
public:
   char name[20];
   char address[20];
};


void ShowN(Internet &obj)//函数定义,不能写成,void Internet::ShowN(Internet &obj)
{
   cout<<obj.name<<endl;
}
void main()
{
   Internet a("中国软件开发实验室","www.cndev-lab.com");
```

```
    ShowN(a);
    cin.get();
}
```

上面的代码通过友元函数的定义，我们成功的访问到了 a 对象的保护成员 name,友元函数并不能看做是类的成员函数，它只是个被声明为类友元的普通函数，所以在类外部函数的定义部分不能够写成 void Internet::ShowN(Internet &obj)，这一点要注意。

一个普通函数可以是多个类的友元函数，对上面的代码我们进行修改，注意观察变化：

```
#include <iostream>
using namespace std;
class Country;
class Internet
{
public:
    Internet(char *name,char *address)
    {
        strcpy(Internet::name,name);
        strcpy(Internet::address,address);
    }
friend void ShowN(Internet &obj,Country &cn);//注意这里
public:
    char name[20];
    char address[20];
};

class Country
{
public:
    Country()
    {
        strcpy(cname,"中国");
    }
friend void ShowN(Internet &obj,Country &cn);//注意这里
protected:
    char cname[30];
};

void ShowN(Internet &obj,Country &cn)
{
    cout<<cn.cname<<"|"<<obj.name<<endl;
}
void main()
{
```

```
    Internet a("中国软件开发实验室","www.cndev-lab.com");
    Country b;
    ShowN(a,b);
    cin.get();
}
```

一个类的成员函数函数也可以是另一个类的友元，从而可以使得一个类的成员函数可以操作另一个类的数据成员，我们在下面的代码中增加一类 Country，注意观察：

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

class Country
{
public:
    Country()
    {
        strcpy(cname,"中国");
    }
    void Editurl(Internet &temp);//成员函数的声明
protected:
    char cname[30];
};
class Internet
{
public:
    Internet(char *name,char *address)
    {
        strcpy(Internet::name,name);
        strcpy(Internet::address,address);
    }
    friend void Country::Editurl(Internet &temp);//友元函数的声明
protected:
    char name[20];
    char address[20];
};
void Country::Editurl(Internet &temp)//成员函数的外部定义
{
    strcpy(temp.address,"edu.cndev-lab.com");
    cout<<temp.name<<"|"<<temp.address<<endl;
}
void main()
{
```

```
    Internet a("中国软件开发实验室","www.cndev-lab.com");
    Country b;
    b.Editurl(a);
    cin.get();
}
```

整个类也可以是另一个类的友元，该友元也可以称做为**友类**。友类的每个成员函数都可以访问另一个类的所有成员。

示例代码如下：

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

class Country
{
public:
    Country()
    {
        strcpy(cname,"中国");
    }
    friend class Internet;//友类的声明
protected:
    char cname[30];
};
class Internet
{
public:
    Internet(char *name,char *address)
    {
        strcpy(Internet::name,name);
        strcpy(Internet::address,address);
    }
    void Editcname(Country &temp);
protected:
    char name[20];
    char address[20];
};
void Internet::Editcname(Country &temp)
{
    strcpy(temp.cname,"中华人民共和国");
}
void main()
{
```

```
    Internet a("中国软件开发实验室","www.cndev-lab.com");
    Country b;
    a.Editcname(b);
    cin.get();
}
```

在上面的代码中我们成功的通过 Internet 类 Editcname 成员函数操作了 Country 类的保护成员 cname。

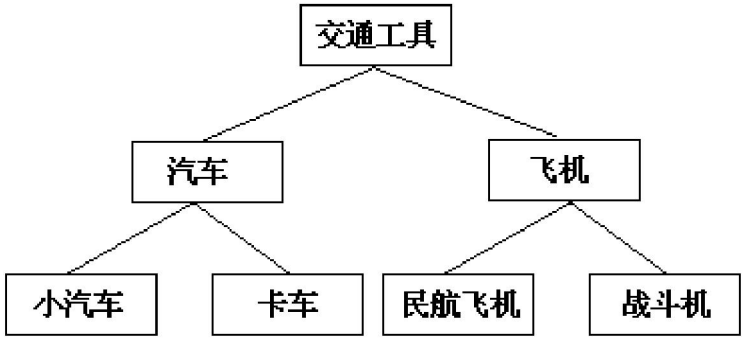
在编程中，我们使用友元的另外一个重要原因是为了方便重载操作符的使用，这些内容我们将在后面的教程着重讨论！

24.图例实解:C++中类的继承特性

整个 c++程序设计全面围绕面向对象的方式进行，**类的继承特性**是 c++的一个非常非常重要的机制，继承特性可以使一个新类获得其父类的操作和数据结构，程序员只需在新类中增加原有类中没有的成分。

可以说这一章节的内容是 c++面向对象程序设计的关键。

下面我们简单的来说一下继承的概念，先看下图:



上图是一个抽象描述的特性继承表

交通工具是一个**基类**（**也称做父类**），通常情况下所有交通工具所共同具备的特性是速度与额定载人的数量，但按照生活常规，我们来继续给交通工具来细分类的时候，我们会分别想到有汽车类和飞机类等等，汽车类和飞类同样具备速度和额定载人数量这样的特性，而这些特性是所有交通工具所共有的，那么当建立汽车类和飞机类的时候我们无需再定义基类已经有的数据成员，而只需要描述汽车类和飞机类所特有的特性即可，飞机类和汽车类的特性是由在交通工具类原有特性基础上增加而来的，那么飞机类和汽车类就是交通工具类的**派生类**（**也称做子类**）。以此类推，层层递增，这种子类获得父类特性的概念就是继承。

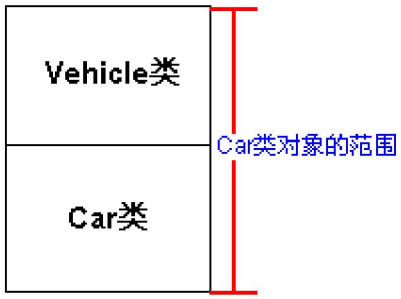
下面我们根据上图的理解，有如下的代码：

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

class Vehicle
{
public:
   void EditSC(float speed,int total);
protected:
   float speed;//速度
   int total;//最大载人量
};
```

```
void Vehicle::EditSC(float speed,int total)
{
   Vehicle::speed = speed;
   Vehicle::total = total;
}
class Car:public Vehicle//Car 类继承 Vehicle 的特性，Car 类是 Vehicle 的派生类
{
public:
   Car()
   {
      aird=0;
   }
protected:
   int aird;//排量
};

class plane:public Vehicle
{
protected:
   float wingspan;//翼展
};

void main()
{
   Car a;
   a.EditSC(150,4);
   cin.get();
}
```

派生类的定义可以在类名称后加冒号 public 空格加基类名称进行定义，如上面代码中的 class Car:public Vehicle。

一旦成功定义派生类，那么派生类就可以操作基类的所有数据成员包括是受保护型的，上面代码中的 a.EditSC(100,4)；就是例子，甚至我们可以在构造派生类对象的时候初始化他们，但我们是不推荐这么做的,因为类于类之间的操作是通过接口进行勾通的，为了不破坏类的这种封专装特性，即使是父类于子类的操作也应按遵循这个思想，这么做的好处也是显而易见的，当基类有错的时候，只要不涉及接口，那么基类的修改就不会影响到派生类的操作。



至于为什么派生类能够对基类成员进行操作，我们上图可以简单的说明基类与子类在内存中的排列状态。

我们知道，类对象操作的时候在内部构造的时候会有一个隐的 this 指针，由于 Car 类是 Vehicle 的派生类，那么当 Car 对象创建的时候，这个 this 指针就会覆盖到 Vehicle 类的范围，所以派生类能够对基类成员进行操作。

笔者写到这里的时候不得不提一下，我有开发 c#与 java 的经验，就这两种语言来说，学到这里的时候很多人很难理解继承这一部分的内容，或者是理解的模糊不清，其实正是缺少了与 this 指针相关的 c++知识，多数高级语言的特性是不涉及内存状态的操作，java 与 c#是接触不到这些知识的，所以理解起这部分内容就更抽象更不具体。

下面我们来说一下,**派生类对象(子类对象)的构造**。

由上面的例程我们知道 Car 类是 Vehicle 类的派生类(子类)，c++规定，创建派生类对象的时候**首先**调用**基类的构造函数**初始化基类成员，**随后**才调用**派生类构造函数**。

但是要注意的是，在创建派生类对象之前，系统首先确定派生类对象的覆盖范围（也可以称做大小尺寸），上面代码的派生类对象 a 就覆盖于 Vehicle 类和 Car 类上，至于派生类对象的创建是如何构造基类成员的，我们看如下代码，随后再进行分析：

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

class Vehicle
{
public:
   Vehicle(float speed=0,int total=0)
   {
      cout<<"载入 Vehicle 类构造函数"<<endl;
      Vehicle::speed = speed;
      Vehicle::total = total;
   }
   Vehicle(Vehicle &temp)
   {
      Vehicle::speed = temp.speed;
      Vehicle::total = temp.total;
   }
```

```
    ~Vehicle()
    {
        cout<<"载入 Vehicle 类析构函数"<<endl;
        cin.get();
    }
protected:
    float speed;//速度
    int total;//最大载人量
};
class Car:public Vehicle
{
public:
    Car(float aird=0,float speed = 0,int total = 0):Vehicle(speed,total)
    {
        cout<<"载入 Car 类构造函数"<<endl;
        Car::aird = aird;
    }
    Car(Car &temp):Vehicle(temp.speed,temp.total)
    {
        cout<<"载入 Car 类拷贝构造函数"<<endl;
        Car::aird = temp.aird;
    }
    ~Car()
    {
        cout<<"载入 Car 类析构函数"<<endl;
        cin.get();
    }
protected:
    float aird;//排量
};
void main()
{
    Car a(250,150,4);
    Car b = a;
    cin.get();
}
```

对象 a 创建的时候通过 Car(float aird = 0, float speed = 0, int total = 0):Vehicle(speed, total)，也就是 Car 类构造函数，来构造 Car 类对象成员，但按照 c++的规定首先应该调用基类构造函数构造基成员，在这里基类成员的构造是通过 Vehicle(speed, total)，来实现的。

但值得注意的是 Vehicle(speed, total)的意义并不是对 Vehicle 类的个个成员的初始化，事实上是利用它创建了一个 Vehicle 类的无名对象，由于 Car 类对象的覆盖范围是覆盖于 Vehicle 类和 Car 类上，所以系统在确定了派生类对象 a 的空间范围后，确定了 this 指

针位置，这个 this 指针指向了 Vehicle 类的那个无名对象,这个成员赋值过程就是，this->speed＝无名对象.speed。

其实这里概念比较模糊，笔者因为个人能力的原因暂时也无法说的更明确了，读者对此处知识点的学习，应该靠自己多对比多练习，进行体会理解。

许多书籍对于派生类对象的复制这一知识点多是空缺的，为了教程的易读性，我还是决定说一下在复制过程中容易出错的地方，Car b=a;是派生类对象复制的语句，通过前面教程的学习我们我们知道，类对象的复制是通过拷贝构造函数来完成的，如果上面的例子我们没有提供拷贝构造函数不完整如下：

```
Car(Car &temp)
{
   cout<<"载入 Car 类拷贝构造函数"<<endl;
   Car::aird = temp.aird;
}
```

那么复制过程中就会丢失基类成员的数据了，所以 Car 类拷贝构造函数不能缺少 Vehicle 类无名对象的创建过程:Vehicle(temp.speed,temp.total)，**派生类对象的复制过程系统是不会再调用基类的拷贝构造函数的**，this 指针的问题再次在这里体现出来，大家可以尝试着把无名对象的创建去掉，观察 b.speed 的变化。

类对象够创建必然就有析构过程，派生类对象的析构过程首先是调用派生类的析构过程，再调用基类的构造函数，正好和创建过程相反，在这里笔者已经在上面代码中加入了过程显示，读者可以自行编译后观察。

最后我们说一下**类的继承与组合**。

其实类的组合我们在早些的前面的教程就已经接触过，只是在这里换了个说法而已，当一个类的成员是另一个类的对象的时候就叫做组合，事实上就是类于类的组合。组合和继承是有明显分别的，为了充分理解继承与组合的关系，我们在不破坏类的封装特性的情况下，先看如下的代码：

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

class Vehicle
{
public:
   Vehicle(float speed=0,int total=0)
   {
      Vehicle::speed = speed;
      Vehicle::total = total;
   }
protected:
   float speed;//速度
   int total;//最大载人量
};
```

```
class Motor
{
public:
    Motor(char *motor)
    {
        Motor::motortype = motor;
    }

    char* SMT(Motor &temp);
protected:
    char *motortype;//发动机型号
};
char* Motor::SMT(Motor &temp)
{
    return temp.motortype;
}

class Car:public Vehicle//继承的体现
{
public:
    Car(float speed,int total,int aird,char *motortype):Vehicle(speed,total),motor(motortype)
    {
        Car::aird = aird;
    }

    Motor rm(Car &temp);
protected:
    int aird;//排量
    Motor motor;//类组合的体现
};

Motor Car::rm(Car &temp)
{
    return temp.motor;
}

//------------------------------------------------------------
void test1(Vehicle &temp)
{
    //中间过程省略
};
void test2(Motor &temp)
{
    cout<<temp.SMT(temp);//读者这里注意一下,temp 既是对象也是对象方法的形参
```

```
}
//-------------------------------------------------------------

void main()
{
   Car a(150,4,250,"奥地利 AVL V8");
   test1(a);
   //test2(a);//错误,Car 类与 Motor 类无任何继承关系
   test2(a.rm(a));//如果 Car 类成员是 public 的那么可以使用 test2(a.motor)
   cin.get();
}
```

在上面的代码中我们新增加了发动机类 Motor，Car 类增加了 Motor 类型的 motor 成员，这就是组合，拥有继承特性的派生类可以操作基类的任何成员，但对于与派生类组合起来的普通类对象来说，它是不会和基类有任何联系的。

函数调用：test1(a);，可以成功执行的原因就是**因为 Car 类对象在系统是看来一个 Vehicle 类对象**，即 **Car 类是 Vehicle 类的一种，Car 类的覆盖范围包含了 Vehicle**。

函数调用：test2(a);，执行错误的原因是因为 Motor 类并不认可 Car 类对象 a 与它有任何关系，但我们可以通过使用 Car 类对象 a 的 Motor 类成员 motor，作为函数形参的方式来调用 test2 函数(test2(a.motor))，在这里由于类的成员是受保护的所以我们利用 a.rm(a)来返回受保护的 motor，作为函数参数进行调用。

25.C++中类的多态与虚函数的使用

类的多态特性是支持面向对象的语言最主要的特性，有过非面向对象语言开发经历的人，通常对这一章节的内容会觉得不习惯，因为很多人错误的认为，支持类的封装的语言就是支持面向对象的，其实不然，Visual BASIC 6.0 是典型的非面向对象的开发语言，但是它的确是支持类，支持类并不能说明就是支持面向对象，能够解决多态问题的语言，才是真正支持面向对象的开发的语言，所以务必提醒有过其它非面向对象语言基础的读者注意！

多态的这个概念稍微有点模糊，如果想在一开始就想用清晰用语言描述它，让读者能够明白，似乎不太现实，所以我们先看如下代码：

```
//例程 1
#include <iostream>
using namespace std;

class Vehicle
{
public:
   Vehicle(float speed,int total)
   {
      Vehicle::speed=speed;
      Vehicle::total=total;
   }
   void ShowMember()
   {
      cout<<speed<<"|"<<total<<endl;
   }
protected:
   float speed;
   int total;
};
class Car:public Vehicle
{
public:
   Car(int aird,float speed,int total):Vehicle(speed,total)
   {
      Car::aird=aird;
   }
   void ShowMember()
```

```
    {
        cout<<speed<<"|"<<total<<"|"<<aird<<endl;
    }
protected:
    int aird;
};

void main()
{
    Vehicle a(120,4);
    a.ShowMember();
    Car b(180,110,4);
    b.ShowMember();
    cin.get();
}
```

在 c++中是允许派生类重载基类成员函数的，对于类的重载来说，明确的，不同类的对象，调用其类的成员函数的时候，系统是知道如何找到其类的同名成员，上面代码中的 a.ShowMember();，即调用的是 Vehicle::ShowMember()，b.ShowMember();，即调用的是 Car::ShowMemeber();。

但是在实际工作中，很可能会碰到对象所属类不清的情况，下面我们来看一下派生类成员作为函数参数传递的例子，代码如下：

```
//例程 2
#include <iostream>
using namespace std;

class Vehicle
{
public:
    Vehicle(float speed,int total)
    {
        Vehicle::speed=speed;
        Vehicle::total=total;
    }
    void ShowMember()
    {
        cout<<speed<<"|"<<total<<endl;
    }
protected:
    float speed;
    int total;
};
class Car:public Vehicle
{
```

```
public:
    Car(int aird,float speed,int total):Vehicle(speed,total)
    {
        Car::aird=aird;
    }
    void ShowMember()
    {
        cout<<speed<<"|"<<total<<"|"<<aird<<endl;
    }
protected:
    int aird;
};

void test(Vehicle &temp)
{
    temp.ShowMember();
}

void main()
{
    Vehicle a(120,4);
    Car b(180,110,4);
    test(a);
    test(b);
    cin.get();
}
```

例子中，对象 a 与 b 分辨是基类和派生类的对象，而函数 test 的形参却只是 Vehicle 类的引用，按照类继承的特点，系统把 Car 类对象看做是一个 Vehicle 类对象，因为 Car 类的覆盖范围包含 Vehicle 类，所以 test 函数的定义并没有错误，我们想利用 test 函数达到的目的是，传递不同类对象的引用，分别调用不同类的,重载了的,ShowMember 成员函数，但是程序的运行结果却出乎人们的意料，系统分不清楚传递过来的基类对象还是派生类对象，无论是基类对象还是派生类对象调用的都是基类的 ShowMember 成员函数。

为了要解决上述不能正确分辨对象类型的问题，c++提供了一种叫做**多态性**（polymorphism）的技术来解决问题，对于例程序 1，这种能够在编译时就能够确定哪个重载的成员函数被调用的情况被称做**先期联编**（early binding），而在系统能够在运行时，能够根据其类型确定调用哪个重载的成员函数的能力，称为**多态性**，或叫**滞后联编**(late binding)，下面我们要看的例程 3，就是滞后联编，滞后联编正是解决多态问题的方法。

代码如下：

```
//例程 3
#include <iostream>
using namespace std;

class Vehicle
```

```
{
public:
    Vehicle(float speed,int total)
    {
        Vehicle::speed = speed;
        Vehicle::total = total;
    }
    virtual void ShowMember()//虚函数
    {
        cout<<speed<<"|"<<total<<endl;
    }
protected:
    float speed;
    int total;
};
class Car:public Vehicle
{
public:
    Car(int aird,float speed,int total):Vehicle(speed,total)
    {
        Car::aird = aird;
    }
    virtual void ShowMember()//虚函数,在派生类中，由于继承的关系，这里的 virtual 也可以不加
    {
        cout<<speed<<"|"<<total<<"|"<<aird<<endl;
    }
public:
    int aird;
};

void test(Vehicle &temp)
{
    temp.ShowMember();
}

int main()
{
    Vehicle a(120,4);
    Car b(180,110,4);
    test(a);
    test(b);
    cin.get();
}
```

多态特性的工作依赖虚函数的定义，在需要解决多态问题的重载成员函数前，加上 virtual 关键字，那么该成员函数就变成了虚函数，从上例代码运行的结果看，系统成功的分辨出了对象的真实类型，成功的调用了各自的重载成员函数。

多态特性让程序员省去了细节的考虑，提高了开发效率，使代码大大的简化，当然虚函数的定义也是有缺陷的，因为多态特性增加了一些数据存储和执行指令的开销，所以能不用多态最好不用。

虚函数的定义要遵循以下重要规则：

1. 如果虚函数在基类与派生类中出现，仅仅是名字相同，而形式参数不同，或者是返回类型不同，那么即使加上了 virtual 关键字，也是不会进行滞后联编的。

2. 只有类的成员函数才能说明为虚函数，因为虚函数仅适合用与有继承关系的类对象，所以普通函数不能说明为虚函数。

3. 静态成员函数不能是虚函数,因为静态成员函数的特点是不受限制于某个对象。

4. 内联(inline)函数不能是虚函数，因为内联函数不能在运行中动态确定位置。即使虚函数在类的内部定义定义，但是在编译的时候系统仍然将它看做是非内联的。

5. 构造函数不能是虚函数，因为构造的时候，对象还是一片位定型的空间，只有构造完成后，对象才是具体类的实例。

6. 析构函数可以是虚函数,而且通常声名为虚函数。

说明一下，虽然我们说使用虚函数会降低效率，但是在处理器速度越来越快的今天，将一个类中的所有成员函数都定义成为 virtual 总是有好处的，它除了会增加一些额外的开销是没有其它坏处的，对于保证类的封装特性是有好处的。

对于上面虚函数使用的重要规则 6，我们有必要用实例说明一下，**为什么具备多态特性的类的析构函数，有必要声明为 virtual**。

代码如下：

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

class Vehicle
{
public:
   Vehicle(float speed,int total)
   {
      Vehicle::speed=speed;
      Vehicle::total=total;
   }
   virtual void ShowMember()
   {
      cout<<speed<<"|"<<total<<endl;
   }
```

```
    virtual ~Vehicle()
    {
        cout<<"载入 Vehicle 基类析构函数"<<endl;
        cin.get();
    }
protected:
    float speed;
    int total;
};
class Car:public Vehicle
{
public:
    Car(int aird,float speed,int total):Vehicle(speed,total)
    {
        Car::aird=aird;
    }
    virtual void ShowMember()
    {
        cout<<speed<<"|"<<total<<"|"<<aird<<endl;
    }
    virtual ~Car()
    {
        cout<<"载入 Car 派生类析构函数"<<endl;
        cin.get();
    }
protected:
    int aird;
};

void test(Vehicle &temp)
{
    temp.ShowMember();
}
void DelPN(Vehicle *temp)
{
    delete temp;
}
void main()
{
    Car *a=new Car(100,1,1);
    a->ShowMember();
    DelPN(a);
    cin.get();
}
```

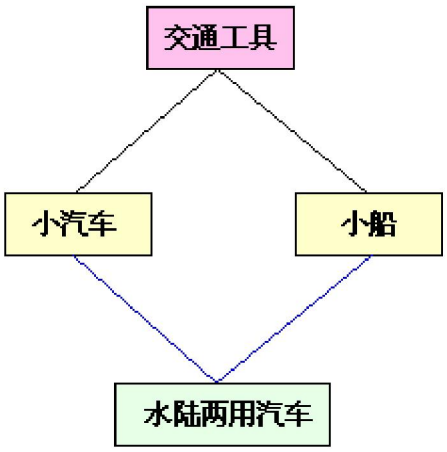
从上例代码的运行结果来看，当调用 DelPN(a);后，在析构的时候，系统成功的确定了先调用 Car 类的析构函数，而如果将析构函数的 virtual 修饰去掉，再观察结果，会发现析构的时候，始终只调用了基类的析构函数，由此我们发现，多态的特性的 virtual 修饰，不单单对基类和派生类的普通成员函数有必要，而且对于基类和派生类的析构函数同样重要。

26.图文例解 C++类的多重继承与虚拟继承

在过去的学习中，我们始终接触的单个类的继承，但是在现实生活中，一些新事物往往会拥有两个或者两个以上事物的属性，为了解决这个问题，C++引入了多重继承的概念，**C++允许为一个派生类指定多个基类，这样的继承结构被称做多重继承**。

举个例子，交通工具类可以派生出汽车和船连个子类，但拥有汽车和船共同特性水陆两用汽车就必须继承来自汽车类与船类的共同属性。

由此我们不难想出如下的图例与代码：



当一个派生类要使用多重继承的时候，必须在派生类名和冒号之后列出所有基类的类名，并用逗好分隔。

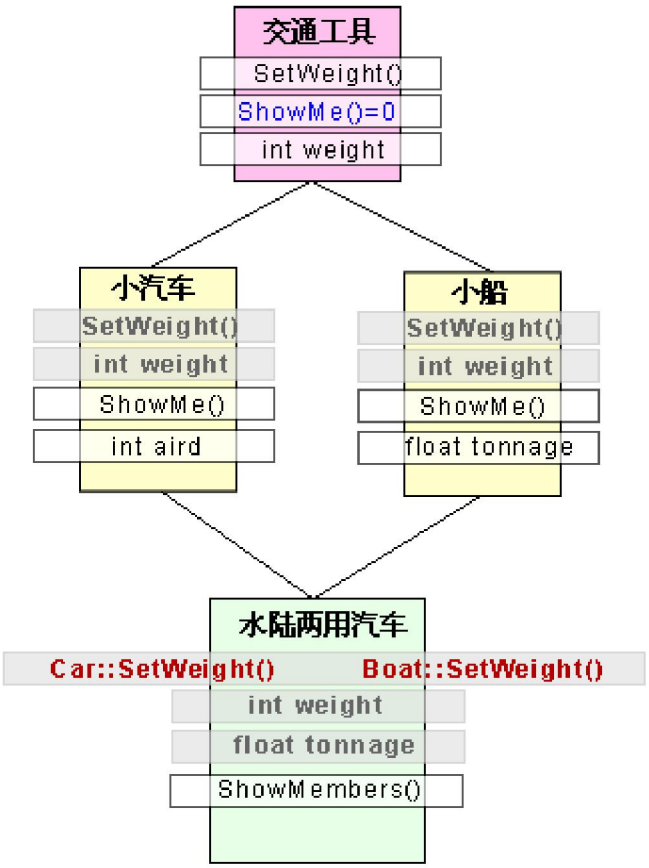
```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Vehicle
{
   public:
      Vehicle(int weight = 0)
      {
         Vehicle::weight = weight;
      }
      void SetWeight(int weight)
```

```
    {
        cout<<"重新设置重量"<<endl;
        Vehicle::weight = weight;
    }
    virtual void ShowMe() = 0;
protected:
    int weight;
};
class Car:public Vehicle//汽车
{
public:
    Car(int weight=0,int aird=0):Vehicle(weight)
    {
        Car::aird = aird;
    }
    void ShowMe()
    {
        cout<<"我是汽车！"<<endl;
    }
protected:
    int aird;
};

class Boat:public Vehicle//船
{
public:
    Boat(int weight=0,float tonnage=0):Vehicle(weight)
    {
        Boat::tonnage = tonnage;
    }
    void ShowMe()
    {
        cout<<"我是船！"<<endl;
    }
protected:
    float tonnage;
};

class AmphibianCar:public Car,public Boat//水陆两用汽车,多重继承的体现
{
public:
    AmphibianCar(int weight,int aird,float tonnage)
    :Vehicle(weight),Car(weight,aird),Boat(weight,tonnage)
    //多重继承要注意调用基类构造函数
```

```
    {


    }
    void ShowMe()
    {
        cout<<"我是水陆两用汽车！"<<endl;
    }
};
int main()
{
   AmphibianCar a(4,200,1.35f);//错误
   a.SetWeight(3);//错误
   system("pause");
}
```

上面的代码从表面看，看不出有明显的语发错误，但是它是不能够通过编译的。这有是为什么呢？
这是由于多重继承带来的继承的模糊性带来的问题。

先看如下的图示：



在图中深红色标记出来的地方正是主要问题所在，水陆两用汽车类继承了来自 Car 类与 Boat 类的属性与方法，Car 类与 Boat 类同为 AmphibianCar 类的基类，在内存分配上 AmphibianCar 获得了来自两个类的 SetWeight()成员函数，当我们调用 a.SetWeight(3)的时候计算机不知道如何选择分别属于两个基类的被重复拥有了的类成员函数 SetWeight()。

由于这种模糊问题的存在同样也导致了 AmphibianCar a(4,200,1.35f);执行失败，系统会产生 Vehicle”不是基或成员的错误。

以上面的代码为例，我们要想让 AmphibianCar 类既获得一个 Vehicle 的拷贝，而且又同时共享用 Car 类与 Boat 类的数据成员与成员函数就必须通过 C++所提供的**虚拟继承**技术来实现。

我们在 Car 类和 Boat 类继承 Vehicle 类出，在前面加上 virtual 关键字就可以实现虚拟继承，使用虚拟继承后，当系统碰到多重继承的时候就会自动先加入一个 Vehicle 的拷贝，当再次请求一个 Vehicle 的拷贝的时候就会被忽略，保证继承类成员函数的唯一性。

修改后的代码如下，注意观察变化：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Vehicle
{
  public:
    Vehicle(int weight = 0)
    {
      Vehicle::weight = weight;
      cout<<"载入 Vehicle 类构造函数"<<endl;
    }
    void SetWeight(int weight)
    {
      cout<<"重新设置重量"<<endl;
      Vehicle::weight = weight;
    }
    virtual void ShowMe() = 0;
  protected:
    int weight;
};
class Car:virtual public Vehicle//汽车，这里是虚拟继承
{
  public:
    Car(int weight=0,int aird=0):Vehicle(weight)
    {
      Car::aird = aird;
```

```
        cout<<"载入 Car 类构造函数"<<endl;
    }
    void ShowMe()
    {
        cout<<"我是汽车！"<<endl;
    }
  protected:
    int aird;
};

class Boat:virtual public Vehicle//船,这里是虚拟继承
{
  public:
    Boat(int weight=0,float tonnage=0):Vehicle(weight)
    {
        Boat::tonnage = tonnage;
        cout<<"载入 Boat 类构造函数"<<endl;
    }
    void ShowMe()
    {
        cout<<"我是船！"<<endl;
    }
  protected:
    float tonnage;
};

class AmphibianCar:public Car,public Boat//水陆两用汽车,多重继承的体现
{
  public:
    AmphibianCar(int weight,int aird,float tonnage)
    :Vehicle(weight),Car(weight,aird),Boat(weight,tonnage)
    //多重继承要注意调用基类构造函数
    {
        cout<<"载入 AmphibianCar 类构造函数"<<endl;
    }
    void ShowMe()
    {
        cout<<"我是水陆两用汽车！"<<endl;
    }
    void ShowMembers()
    {
        cout<<"重量："<<weight<<"顿，"<<"空气排量："<<aird<<"CC，"<<"排水量："<<tonnage<<"顿"<<endl;
    }
};
```

```
int main()
{
    AmphibianCar a(4,200,1.35f);
    a.ShowMe();
    a.ShowMembers();
    a.SetWeight(3);
    a.ShowMembers();
    system("pause");
}
```

注意观察类构造函数的构造顺序。

虽然说虚拟继承与虚函数有一定相似的地方，但读者务必要记住，他们之间是绝对没有任何联系的！

27.类的分解，抽象类与纯虚函数的需要性

为了不模糊概念在这里我们就简单的阐述一下**类的分解**，前面的教程我们着重讲述了类的继承，继承的特点就是，派生类继承基类的特性，进行结构扩张，这种逐步扩张，逐步在各派生类中分解彼此不同特性的过程其实就是类的分解。

分解过程笔者在这里不想再拿代码进行过多阐述分析了，意思说到，对于逐步分解，逐步扩张的思想就靠大家自己思考了。

拿前面交通工具类的程序进行思考，由交通工具派生出来的汽车类，飞机类，是具备更具体特性的描述的类，而对于交通工具这一个基类来说，它的特性是模糊的，广泛的，如果建立一个交通工具类的对象并没有实际意义，为了对这种没有必要能够建立对象的类进行约束，c++引入了**抽象类**的特性，抽象类的约束控制来源于纯虚函数的定义。

生命一个类的成员函数为纯虚函数的意义在于让c++知道该函数并无意义，它的作用只是为派生类进行虚函数重载保留位置。

纯虚函数的定义方法就是在类的成员函数的声明之后加上“=0”的标记，类中一旦有纯虚函数的定义那么这个类就再也不能创建此类的对象了，我们把这种类叫做抽象类。

抽象类的示例代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必注明出处和作者
#include <iostream>
using namespace std;

class Vehicle
{
public:
   Vehicle(float speed,int total)
   {
      Vehicle::speed = speed;
      Vehicle::total = total;
   }
   virtual void ShowMember()=0;//纯虚函数的定义
protected:
   float speed;
   int total;
};
class Car:public Vehicle
{
public:
```

```
    Car(int aird,float speed,int total):Vehicle(speed,total)
    {
        Car::aird = aird;
    }
    virtual void ShowMember()//派生类成员函数重载
    {
        cout<<speed<<"|"<<total<<"|"<<aird<<endl;
    }
protected:
    int aird;
};

int main()
{
    //Vehicle a(100,4);//错误,抽象类不能创建对象
    Car b(250,150,4);
    b.ShowMember();
    system("pause");
}
```

28.C++类的继承与多重继承的访问控制

在前面的练习中我们一直在使用 public 的继承方式，即共有继承方式，对于 protected 和 private 继承方式，即**保护继承**与**私有继承**方式我们并没有讨论。

对于单个类来说，讨论保护继承与私有继承的区别意义是不大的，他们的区别只在多级继承的情况中体现。

在这里我声明一下,对于此章节的内容不太适合用过多的文字进行描述，主要还是看例子，通过例子熟悉之间的关系，过多的文字描述会模糊读者思路。

例程如下（重要部分都做了详细说明）：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Base
{
   public://公有的
      int a1;
      virtual void test() = 0;
   protected://受保护的
      int a2;
   private://私有的
      int a3;
};
//-------------------------------------------------------------------------
class ProtectedClass:protected Base//保护继承
{
   public:
      void test()
      {
         a1 = 1;//a1 在这里被转变为 protected
         a2 = 2;//a2 在这里被转变为 protected
         //a3=3;//错误，派生类不能访问基类的私有成员
      }
};
class ControlProtectedClass:public ProtectedClass//以 public 方式继承 ProtectedClass 类
```

```
{
  public:
    void test()
    {
      a1 = 1;//a1 在这里仍然保持为 a1 在这里被转变为 protected
      a2 = 2;//a2 在这里仍然保持为 a1 在这里被转变为 protected
      //a3=3;//错误,由于 Base 类成员为私有的，即使是上级父类是保护继承，也不能改变 Base 类成员的控制类型
    }
};
//-------------------------------------------------------------------------
class PrivateClass:private Base//私有继承
{
  public:
    void test()
    {
      a1 = 1;//a1 在这里被转变为 private
      a2 = 2;//a2 在这里被转变为 private
      //a3=3;//错误，基类私有成员对文件区域与派生类区域都是不可访问的
    }
};
class ControlPrivateClass:public PrivateClass//以 public 方式继承 PrivateClass 类
{
  public:
    void test()
    {
      //a1=1;//错误，由于基类 PrivateClass 为私有继承,a1 已经转变为 private
      //a2=2;//错误，由于基类 PrivateClass 为私有继承,a1 已经转变为 private
      //a3=3;//错误,由于 Base 类成员为私有的，PrivateClass 类也为私有继承
    }
};
//-------------------------------------------------------------------------
class PublicClass:public Base//共有继承有区别与其它方式的继承，继承后的各成员不会其改变控制方式
{
  public:
    void test()
    {
      a1 = 1;//a1 仍然保持 public
      a2 = 2;//a2 仍然保持 protected
      //a3=3;//错误，派生类不能操作基类的私有成员
    }
};
class ControlPublicClass:public PublicClass//以 public 方式继承 PublicClass 类
{
  public:
```

```
    void test()
    {
        a1 = 1;//a1 仍然保持 public
        a2 = 2;//a2 仍然保持 protected
        //a3=3;//错误，由于 Base 类成员为私有成员，即使是上级父类是公有继承，也不能改变 Base 类成员的控制类型
    }
};
//-------------------------------------------------------------------------
int main()
{
   system("pause");
}
```

认真看完了例子，相信细心的读者对于共有继承、保护继承与私有继承的区别与特点已经了解，最后再提醒一下读者，在继承关系中，基类的 private 成员不但对应用程序隐藏，即使是派生类也是隐藏不可访问的，而基类的保护成员只对应用程序隐藏，对于派生类来说是不隐藏的，保护继承与私有继承在实际编程工作中使用是极其少见的，他们只在技术理论上有意义。

29.C++运算符重载函数基础及其值返回状态

运算符重载是C++的重要组成部分，它可以让程序更加的简单易懂，简单的运算符使用可以使复杂函数的理解更直观。

对于普通对象来说我们很自然的会频繁使用算数运算符让他们参与计算，但是对于自定义类的对象来说，我们是无论如何也不能阻止写出像下面的代码一样的程序来的。

例子如下：

```
class Test
{
   //过程省略
}

int main()
{
   Test a,c;
   c=a+a;
}
```

当然这样的代码是不能够通过编译的，c++对自定类的算术运算部分保留给了程序员，这也是符合c++灵活特性的。

在c++中要想实现这样的运算就必须自定义**运算符重载函数**，让它来完整具体工作。

在这里要提醒读者的是，自定义类的运算符重载函数也是函数，你重载的一切运算符不会因为是你自己定义的就改变其运算的优先级，自定义运算符的运算优先级同样遵循与内部运算符一样的顺序。

除此之外，c++也规定了一些运算符不能够自定义重载，例如.、::、.*、.->、?:。

下面我们来学习如何重载运算符，运算符重载函数的形式是：

```
返回类型 operator 运算符符号（参数说明）
{
//函数体的内部实现
}
```

运算符重载函数的使用主要分为两种形式，**一种是作为类的友元函数进行使用，另一种则是作为类的成员函数进行使用**。

下面我们先看一下作为类的友元函数使用的例子：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
   public:
      Test(int a = 0)
      {
         Test::a = a;
      }
      friend Test operator +(Test&,Test&);
      friend Test& operator ++(Test&);
   public:
      int a;
};
Test operator +(Test& temp1,Test& temp2)//+运算符重载函数
{
   //cout<<temp1.a<<"|"<<temp2.a<<endl;//在这里可以观察传递过来的引用对象的成员分量
   Test result(temp1.a+temp2.a);
   return result;
}
Test& operator ++(Test& temp)//++运算符重载函数
{
   temp.a++;
   return temp;
}
int main()
{
   Test a(100);
   Test c=a+a;
   cout<<c.a<<endl;
   c++;
   cout<<c.a<<endl;
   system("pause");
}
```

在例子中，我们对于自定义类 Test 来说，重载了加运算符与自动递增运算符，重载的运算符完成了同类型对象的加运算和递增运算

过程。

重载运算符函数返回类型和形式参数也是根据需要量进行调整的，下面我们来看一下修改后的加运算符重载函数。

代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
   public:
      Test(int a = 0)
      {
         Test::a = a;
      }
      friend Test operator +(Test&,const int&);
   public:
      int a;
};
Test operator +(Test& temp1,const int& temp2)//+运算符重载函数
{
   Test result(temp1.a * temp2);
   return result;
}
int main()
{
   Test a(100);
   Test c = a + 10;
   cout<<c.a<<endl;
   system("pause");
}
```

上面修改后的例子中，我们让重载后的加运算符做的事情，事实上并不是同类型对象的加运算，而是自定义类对象与内置 int 常量对象的乘法运算。

值得注意的是，对于运算符重载来说，我们并不一定要用它一定要做同类型对象的加法或者是其它运算，运算符重载函数本身就是函数，那么在函数体内部我们是可以做任何事情的，但是从不违背常规思维的角度来说，我们没有必要让重载加运算的函数来做与其重载的符号意义上完全不相符的工作，所以在使用重载运算符脱离原意之前，必须保证有足够的理由。

下面我们讨论一下作为类成员函数的运算符重载函数的使用，及其函数的值返回与引用返回的差别。

下面我们先看实例，而后逐步分析。

代码如下(重要部分做了详细的注解)：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
  public:
    Test(int a = 0)
    {
      Test::a = a;
    }
    Test(Test &temp)
    //运算符重载函数为值返回的时候会产生临时变量，临时变量与局部变量 result 的复制会调用拷贝构造函数，临时变量的生命周期是在拷贝构造函数运行完成后才结束，但如果运算符重载函数返回的是引用，那么不会产生临时变量，而局部变量 result 的生命周期在运算符重载函数退出后立即消失，它的生命周期要比临时变量短，所以当外部对象获取返回值的内存地址所存储的值的时候，获得是一个已经失去效果的内存地址中的值，在这里的值返回与引用返回的对比，证明了临时变量的生命周期比局部变量的生命周期稍长。
    {
      cout<<"载入拷贝构造函数"<<"|"<<temp.a<<endl;//注意这里，如果修改运算符重载函数为返回引用，这里就会出现异常，temp.a 将获得一个随机值。
      Test::a = temp.a;
    }
    ~Test()//在 mian()内析构的过程是 result 局部变量产生的
    {
      cout<<"载入析构函数!"<<endl;
      cin.get();
    }
    Test operator +(Test& temp2)//+运算符重载函数
    {
      //cout<<this->a<<endl;
      Test result(this->a+temp2.a);
      return result;
    }
    Test& operator ++()//++运算符重载函数

    //递增运算符是单目运算符，使用返回引用的运算符重载函数道理就在于它需要改变自身。
```

```
    //在前面我们学习引用的单元中我们知道，返回引用的函数是可以作为左值参与运算的，这一点也符合单目运算符的特点。
    //如果把该函数改成返回值，而不是返回引用的话就破坏了单目预算改变自身的特点，程序中的++(++c)运算结束后输出 c.a，会发现对象 c 只做了一次递增运算，原因在于，当函数是值返回状态的时候括号内的++c 返回的不是 c 本身而是临时变量，用临时变量参与括号外的++运算，当然 c 的值也就只改变了一次。
    {
        this->a++;
        return *this;
    }
public:
    int a;
};

int main()
{
    Test a(100);
    Test c=a+a;
    cout<<c.a<<endl;
    c++;
    cout<<c.a<<endl;
    ++c;
    cout<<c.a<<endl;
    ++(++c);
    cout<<c.a<<endl;
    system("pause");
}
```

上例中运算符重载函数以类的成员函数方式出现，细心的读者会发现加运算和递增运算重载函数少了一个参数，这是为什么呢？因为当运算符重载函数以类成员函数身份出现的时候，C++会隐藏第一个参数，转而取代的是一个 this 指针。

接下来我们具体分析一下运算符重载函数的值返回与引用返回的差别。

当我们把代码中的加运算重载函数修改成返回引用的时候：

```
    Test& operator +(Test& temp2)//+运算符重载函数
    {
        Test result(this->a+temp2.a);
        return result;
    }
```

执行运算符重载函数返回引用将不产生临时变量，外部的 Test c=a+a; 将获得一个局部的，栈空间内存地址位置上的值，而栈空间的特性告诉我们，当函数退出的时候函数体中局部对象的生命周期随之结束，所以保存在该地址中的数据也将消失，当 c 对象去获取存储在这个地址中的值的时候，里面的数据已经不存在，导致 c 获得的是一个随机值，所以作为双目运算的加运算符重载函数是不益采用返回引用方式编写的，当然如果一定要返回引用，我们可以在堆内存中动态开辟空间存储数据，但是这么做会导致额外的系统开销，同时也会让程序更难读懂。

对于递增运算符来说，它的意义在于能够改变自身，返回引用的函数是可以作为左值参与运算的，所以作为单目运算符，重载它的函数采用返回引用的方式编写是最合适的。

如果我们修改递增运算符重载函数为值返回状态的时候，又会出现什么奇怪的现象呢？

代码如下：

```
Test operator ++()
{
   return this->a++;
}
```

表面上是发现不出什么特别明显的问题的，但是在 main()函数中++(++c);的执行结果却出乎意料，理论上应该是 204 的值，却只是 203，这是为什么呢？

因为当函数是值返回状态的时候括号内的++c 返回的不是 c 本身而是临时变量，用临时变量参与括号外的++运算，当然 c 的值也就只改变了一次。结果为 203 而不是 204。

对于运算符重载函数来说，最后我们还要注意一个问题，**当运算符重载函数的形式参数类型全部为内部类型的时候，将不能重载**。

30.C++中利用构造函数与无名对象简化运算符重载函数

在完整描述思想之前, 我们先看一下如下的例子, 这个例子中的加运算符重载是以非成员函数的方式出现的：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
   public:
      Test(int a)
      {
         Test::a = a;
      }
      friend Test operator + (Test&,int);
   public:
      int a;
};
Test operator + (Test &temp1,int temp2)
{
   Test result(temp1.a + temp2);
   return result;
}
int main()
{
   Test a(100);
   a = a + 10;//正确
   a = 10 + a;//错误
   cout<<a.a<<endl;
   system("pause");
}
```

上面的代码是一个自定义类对象与内置整型对象相加的例子，但错误行让我们猛然感觉很诧异，但仔细看看的确也在情理中，参数顺序改变后 c++无法识别可供使用的运算符重载函数了。

我们为了适应顺序问题不得不多加一个几乎一样的运算符重载函数。

代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
   public:
      Test(int a)
      {
         Test::a = a;
      }
      friend Test operator + (Test&,int);
      friend inline Test operator + (Test&,int);
   public:
      int a;
};
Test operator + (Test &temp1,int temp2)
{
   Test result(temp1.a + temp2);
   return result;
}
inline Test operator + (int temp1,Test &temp2)//利用内联函数的定义提高效率
{
   return temp2+temp1;
}
int main()
{
   Test a(100);
   a = a + 10;//正确
   a = 10 + a;//正确
   cout<<a.a<<endl;
   system("pause");
}
```

代码中我们使用内联函数的目的是为了缩减开销，但事实上我们仍然觉得是比较麻烦的，例子中的情况都还是非成员函数的情况，如果运算符重载函数是作为类成员函数，那么问题就来了，重载函数的第一个参数始终被隐藏，我们无发让 int 形参排列在隐藏参数的

前面，从而导致 a = 10 + a;无法获取正确的运算符重载函数。

有问题的代码如下：

```
class Test
{
  public:
    Test(int a)
    {
      Test::a = a;
    }
    Test operator + (int temp2)
    {
        Test result(temp1.a + temp2);
        return result;
    }
    Test operator + ()//第一个参数被隐藏,怎么办????,int 形参无法放到 this 指针的前面,理想中的应该是(int temp1,Test *this)
    {

    }
  public:
    int a;
};
```

对于这个问题难道没有办法解决吗？

答案是否定的，**我们可以利用类构造函数对参与运算的整型对象进行显式的类型转换，从而生成无名对象参与同类型对象的加运算**，这样做可以缩减代码量，提高程序的可读性。

代码如下（例一为非成员形式，例二为成员形式）：

```
//例一

//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
  public:
    Test(int a)//事实上构造函数起的转换作用本质就是产生无名对象
```

```
    {
        Test::a = a;
    }
    friend Test operator + (Test&,Test&);
    public:
    int a;
};
Test operator + (Test &temp1,Test &temp2)
{
    Test result(temp1.a + temp2.a);
    return result;
}
int main()
{
    Test a(100);
    a = a + Test(10);//显式转换,产生无名对象
    a = Test(10) + a;
    cout<<a.a<<endl;
    a = 50 + 1;//先进行 50+1 的内置整型的加运算，然后进行 a=Test(51)的隐式转换
    cout<<a.a<<endl;
    system("pause");
}
//例二

//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
    public:
        Test(int a)//事实上构造函数起的转换作用本质就是产生无名对象
        {
            Test::a = a;
        }
        Test operator + (Test &temp)//第一个参数即使隐藏也没有关系,因为是以 Test 类型的无名对象参与运算的
        {
            Test result(this->a + temp.a);
            return result;
        }
    public:
```

```
    int a;
};

int main()
{
    Test a(100);
    a = a + Test(10);
    a = Test(10) + a;
    cout<<a.a<<endl;
    a = 50 + 1;//先进行 50+1 的内置整型的加运算，然后进行 a=Test(51)的隐式转换
    cout<<a.a<<endl;
    system("pause");
}
```

认真观察了上面的两个例子后我们可以发现，类的构造函数起了显式或者隐式转换的作用，转换过程实质是产生一个类的无名对象，类的运算符重载函数的参数就是这个无名对象的引用，所以参数的顺序也不再是问题，代码的运行效率也得到提高，无需再定义只是参数顺序不同，内容重复的运算符重载函数了。

31.对 C++递增(增量)运算符重载的思考

在前面的章节中我们已经接触过递增运算符的重载，那时候我们并没有区分前递增与后递增的差别，在通常情况下我们是分别不出++a 与 a++的差别的，但的确他们直接是存在明显差别的。

先看如下代码：

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

int main()
{
   int a=0;
   ++(++a);//正确,(++a)返回的是左值
   (a++)++;//错误,(a++)返回的不是左值
   system("pause");
}
```

代码中(a++)++编译出错误，返回“++”需要左值的错误，这正是前递增与后递增的差别导致的，那么又是为什么呢？

原因主要是由 C++对递增(增量)运算符的定义引发的。

他们之间的差别主要为以下两点：

1、运算过程中，先将对象进行递增修改，而后返回该对象（其实就是对象的引用）的叫**前递增(增量)运算**。在运算符重载函数中采用返回对象引用的方式编写。

2、运算过程中，先返回原有对象的值，而后进行对象递增运算的叫**后递增(增量)运算**。在运算符重载函数中采用值返回的方式编写（这也正是前面(a++)++出错误的原因，(a++)返回的不是引用，不能当作左值继续参加扩号外部的++运算），重载函数的内部实现必须创建一个用于临时存储原有对象值的对象，函数返回的时候就是返回该临时对象。

那么在编写运算符重载函数的时候我们该如何区分前递增运算符重载函数与后递增运算符重载函数呢？

方法就是：在后递增运算符重载函数的参数中多加如一个 int 标识，标记为后递增运算符重载函数。

具体见如下实例（例一为非成员方式，例二为成员方式）：

```
//例一

//程序作者:管宁
```

```
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
   public:
      Test(int a=0)
      {
         Test::a = a;
      }
   friend Test& operator ++ (Test&);
   friend Test operator ++ (Test&,int);
   public:
   int a;
};
Test& operator ++ (Test &temp)//前递增
{
   temp.a++;
   return temp;
}
Test operator ++ (Test &temp,int)//后递增,int 在这里只起到区分作用,事实上并没有实际作用
{
   Test rtemp(temp);//这里会调用拷贝构造函数进行对象的复制工作
   temp.a++;
   return rtemp;
}
int main()
{
   Test a(100);
   ++(++a);
   cout<<a.a<<endl;
   cout<<"观察后递增情况下临时存储对象的值状态："<<(a++).a<<endl;//这里正是体现后递增操作先返回原有对象值地方
   cout<<a.a<<endl;
   (a++)++;
   cout<<a.a<<endl;//由于后递增是值返回状态，所以(a++)++只对 a 做了一次递增操作，操作后为 104 而非 105。
   system("pause");
}
//例二

//程序作者:管宁
//站点:www.cndev-lab.com
```

```
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
   public:
      Test(int a=0)
      {
         Test::a = a;
      }
   Test& operator ++ ();
   Test operator ++ (int);
   public:
   int a;
};
Test& Test::operator ++ ()//前递增
{
   this->a++;
   return *this;
}
Test Test::operator ++ (int)//后递增
{
   Test rtemp(*this);//这里会调用拷贝构造函数进行对象的复制工作

   this->a++;
   return rtemp;
}
int main()
{
   Test a(100);
   ++(++a);
   cout<<a.a<<endl;
   cout<<"观察后递增情况下临时存储对象的值状态："<<(a++).a<<endl;//这里正是体现后递增操作先返回原有对象值地方
   cout<<a.a<<endl;
   (a++)++;
   cout<<a.a<<endl;//由于后递增是值返回状态，所以(a++)++只对 a 做了一次递增操作，操作后为 104 而非 105。
   system("pause");
}
```

通过对前后递增运算的分析，我们可以进一步可以了解到，对于相同情况的单目运算符重载我们都必须做好这些区别工作，保证重载后的运算符符合要求。

32.C++运算符重载转换运算符

为什么需要转换运算符？

大家知道对于内置类型的数据我们可以通过强制转换符的使用来转换数据，例如(int)2.1f;自定义类也是类型，那么自定义类的对象在很多情况下也需要支持此操作，C++提供了转换运算符重载函数,它使得自定义类对象的强转换成为可能。

转换运算符的生命方式比较特别，方法如下：

```
operator 类名();
```

转换运算符的重载函数是**没有返回类型**的，它和类的构造函数，析构函数一样是不遵循函数有返回类型的规定的，**他们都没有返回值**。

下面我看一个例子，看看它是如何工作的：

```
//例 1

//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
  public:
    Test(int a = 0)
    {
      cout<<this<<"："<<"载入构造函数!"<<a<<endl;
      Test::a = a;
    }
    Test(Test &temp)
    {
      cout<<"载入拷贝构造函数!"<<endl;
      Test::a = temp.a;
    }
    ~Test()
    {
```

```
        cout<<this<<"："<<"载入析构函数!"<<this->a<<endl;
        cin.get();
    }
    operator int()//转换运算符
    {
        cout<<this<<"："<<"载入转换运算符函数!"<<this->a<<endl;
        return Test::a;
    }
public:
    int a;
};
int main()
{
    Test b(99);
    cout<<"b 的内存地址"<<&b<<endl;
    cout<<(int)b<<endl;//强转换
    system("pause");
}
```

在例子中我们利用转换运算符将 Test 类的对象强转换成了 int 类型并输出，注意观察转换运算符函数的运行状态，发现并没有产生临时对象，证明了它与普通函数并不相同，虽然它带有 return 语句。

在很多情况下，类的强转换运算符还可以作为类对象加运算重载函数使用，尽管他们的意义并不相同，下面的例子，就是利用转换运算符，将两个类对象转换成 int 后，相加并创建临时类对象，后再赋给另一个对象。

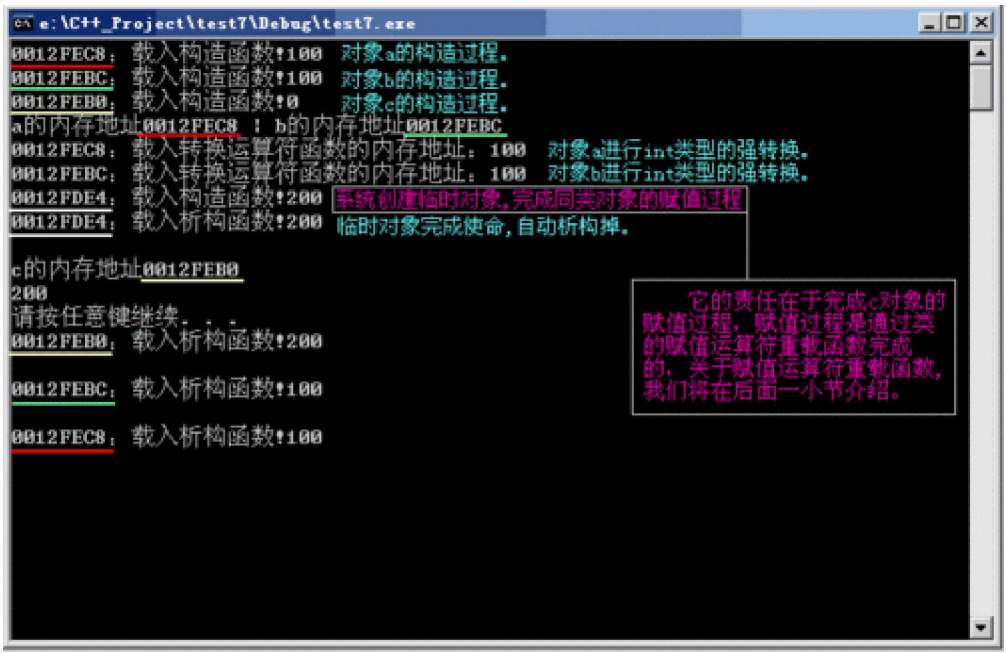
代码如下：

```
//例 2

//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
public:
    Test(int a = 0)
    {
        cout<<this<<"："<<"载入构造函数!"<<a<<endl;
        Test::a = a;
    }
```

```
    Test(Test &temp)
    {
        cout<<"载入拷贝构造函数!"<<endl;
        Test::a = temp.a;
    }
    ~Test()
    {
        cout<<this<<"："<<"载入析构函数!"<<this->a<<endl;
        cin.get();
    }
    operator int()
    {
        cout<<this<<"："<<"载入转换运算符函数的内存地址："<<this->a<<endl;
        return Test::a;
    }
  public:
  int a;
};
int main()
{
  Test a(100),b(100),c;
  cout<<"a 的内存地址"<<&a<<" | b 的内存地址"<<&b<<endl;
  c=Test((int)a+(int)b);//显示式转换
  //c=a+b;//隐式转换
  cout<<"c 的内存地址"<<&c<<endl;
  cout<<c.a<<endl;
  system("pause");
}
```

代码中的 c=a+b;属于隐式转换，它的实现过程与 c=Test((int)a+(int)b);完全相同。

运行结果如下图示（注意观察内存地址，观察构造与析构过程，执行过程图中有解释）：



当一个类含有转换运算符重载函数的时候，有时候会破坏 C++原有规则，导致运算效率降低，这一点不得不注意。

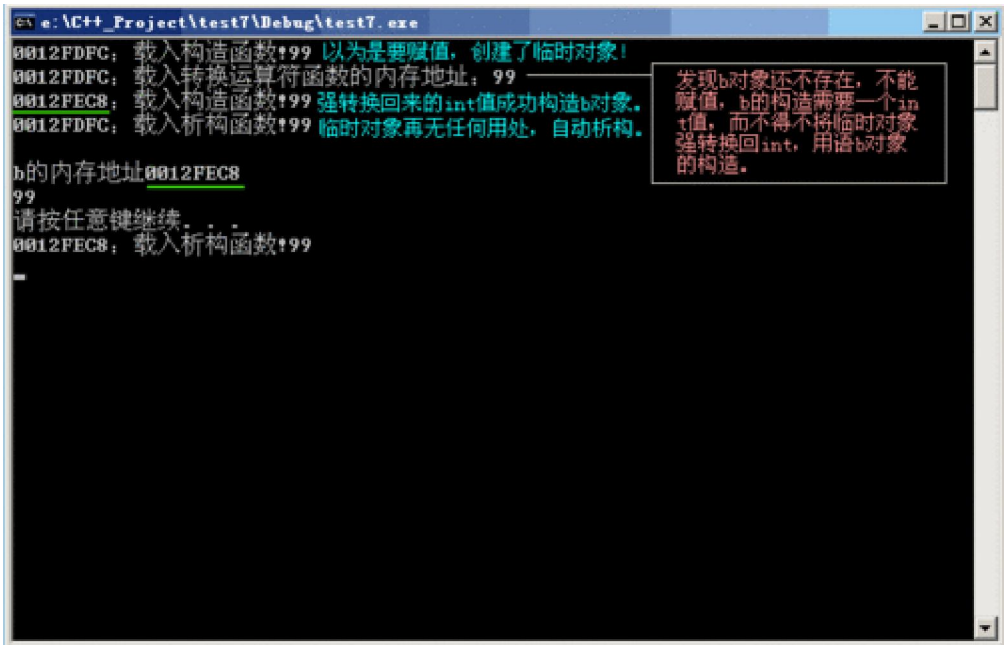
示例如下：

```
//例 3

//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
  public:
    Test(int a = 0)
    {
      cout<<this<<"："<<"载入构造函数!"<<a<<endl;
      Test::a = a;
    }
    Test(Test &temp)
    {
      cout<<"载入拷贝构造函数!"<<endl;
      Test::a = temp.a;
    }
    ~Test()
```

```
    {
        cout<<this<<"："<<"载入析构函数!"<<this->a<<endl;
        cin.get();
    }
    operator int()//转换运算符,去掉则不会调用
    {
        cout<<this<<"："<<"载入转换运算符函数的内存地址："<<this->a<<endl;
        return Test::a;
    }
  public:
  int a;
};
int main()
{
  Test b=Test(99);//注意这里
  cout<<"b 的内存地址"<<&b<<endl;
  cout<<b.a<<endl;
  system("pause");
}
```

按照 C++对无名对象的约定，Test b=Test(99);C++是会按照 Test b(99);来处理的，可是由于转换运算符的加入，导致这一规律被破坏，系统会“错误的”认为你是要给对象赋值，所以系统首先利用 Test(99)创建一个临时对象用于赋值过程使用，可是恰恰系统又没有使用自动提供的赋值运算重载函数去处理，因为发现 b 对象并未构造，转而又不得不将开始原本用于赋值而创建的临时对象再次的强转换为 int 类型，提供给 b 对象进行构造，可见中间的创建临时对象和载入转换运算符函数的过程完全是多余，读者对此例要认真解读，充分理解。

运行结果如下图所示（运行过程的解释见图）：



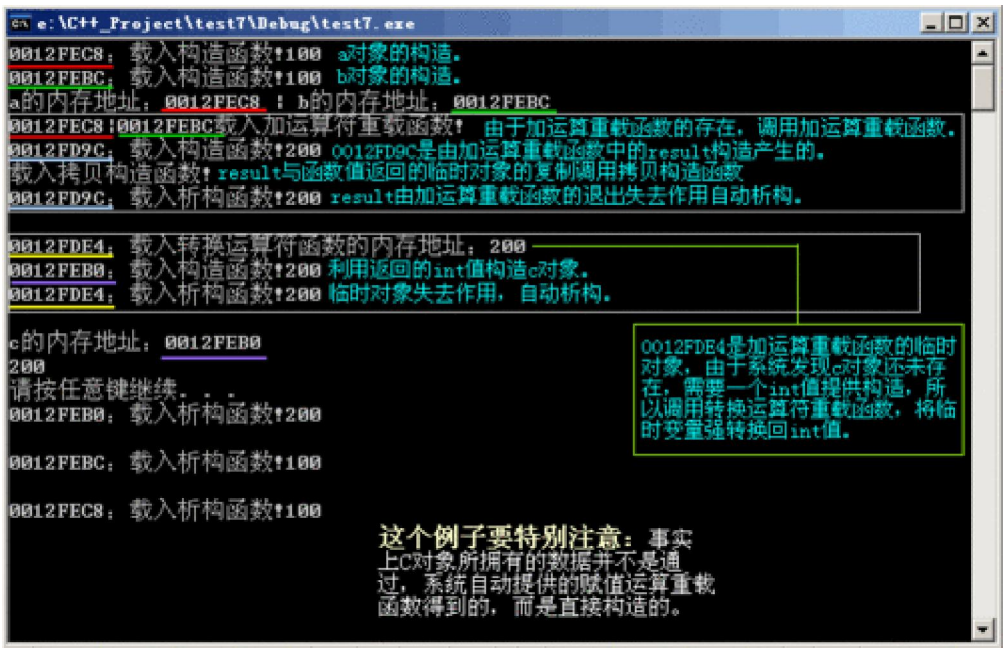
由于类的转换运算符与类的运算符重载函数，在某些地方上使用的时候，有功能相似的地方，如果两者都存在于类中，那么虽然运行结果正确，但其运行过程会出现一些意向不到的步骤，导致程序运行效率降低。

下面的例子就是这个情况，代码如下：

```
//例 4

//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
  public:
    Test(int a = 0)
    {
      cout<<this<<": "<<"载入构造函数!"<<a<<endl;
      Test::a = a;
    }
    Test(Test &temp)
    {
      cout<<"载入拷贝构造函数!"<<endl;
      Test::a = temp.a;
    }
```

```
    ~Test()
    {
        cout<<this<<"："<<"载入析构函数!"<<this->a<<endl;
        cin.get();
    }
    Test operator +(Test& temp2)
    {
        cout<<this<<"|"<<&temp2<<"载入加运算符重载函数!"<<endl;
        Test result(this->a+temp2.a);
        return result;
    }
    operator int()
    {
        cout<<this<<"："<<"载入转换运算符函数的内存地址："<<this->a<<endl;
        return Test::a;
    }
  public:
  int a;
};
int main()
{
  Test a(100),b(100);
  cout<<"a 的内存地址："<<&a<<" | b 的内存地址："<<&b<<endl;
  Test c=a+b;
  cout<<"c 的内存地址："<<&c<<endl;
  cout<<c.a<<endl;
  system("pause");
}
```

运行过程见下图。



从图中我们可以清晰的看到，不必要的运算过程被执行，导致开销增大，读者在理解此例的时候要格外小心！

现在总结一下转换运算符的优点与缺点：

优点：在不提供带有类对象参数的运算符重载函数的情况下，转换运算符重载函数可以将类对象转换成需要的类型，然后进行运算，最后在构造成类对象，这一点和类的运算符重载函数有相同的功效。（例 2 就是这种情况）

缺点：如果一个类只有转换运算符重载函数，而没有真正意义上运算符重载函数，当用转换运算符重载函数替代运算符重载函数，进行工作的时候，就会让程序的可读性降低，歪曲了运算符操作的真正含义。（例 2 中的 c=a+b;//隐式转换，就是例子，事实上 a+b 的作用只是对返回的整型数据进行了加运算，而对象赋值的操作是系统隐式的帮大家转换成了 c=Test(a+b)。）

最后我们来说一下，**多路径转换的多义性问题**，多义性问题一直是 C++编程中容易忽视的问题，但它的确是不容小视，当问题隐藏起来的时候你不会发觉，一旦触发麻烦就来了。

类的**转换构造函数**与类的**转换运算符重载函数**是**互逆**的。（例 3 中的 Test(int a = 0)是将 int 类型的数据转换构造成 Test 类对象，而 operator int()则是将 Test 类对象转换成 int 类型数据）

但是当他们是出现在两个不同的类中，对于一个类对象转换来说，同时拥有两种近似的转换途径的时候，多义性的问题就暴露出来，导致编译出错。

下例就是这个状态：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
```

```
#include <iostream>
using namespace std;
class B;
class A
{
   public:
      A(B &);//转换构造函数,他的作用是用 B 类对象构造 A 类对象
      void Edita(int temp)
      {
         A::a=temp;
      }
   public:
      int a;
};
class B
{
   public:
      B(int a=0)
      {
         B::a=a;
      }
      int Ra()
      {
         return B::a;
      }
      operator A()//转换运算符重载函数,他的作用则是将 B 类对象转换成 A 类对象
      {
         return *this;
      }
   protected:
      int a;
};
A::A(B &temp)
{
   cout<<this<<"|"<<&temp<<endl;
   A::a=temp.Ra();
}
void tp(A temp)
{

}
int main()
{
   B bt(100);
```

```
    A at=A(bt);
    //tp(bt);//错误,多义性问题,系统不知道如何选择,是选择 A(B &)转化构造好呢?还是选择 B::operator A()进行转换好呢?
    tp(A::A(bt));//显示的处理可以消除多义性问题
    system("pause");
}
```

代码中的 A at=A(bt);运行正常，因为系统发现对象 at 还未构造，所以优先选取了 A 类的转换构造函数处理了，没有产生二义性问题。

但是代码中的 tp(bt);编译错误，这是因为函数 tp 的参数要求的是一个 A 类对象，而我们给他的则是一个 B 类对象，而在 A 类与 B 类中都有一个类似的操作，可以将 B 类对象转换成 A 类对象，系统不知道是选取 A 类的转换构造函数进行构造处理，还是选择 B 类中的转换运算符号重载函数处理，系统拒绝从他们两个中选一个，所以编译错误。

我们修改 tp(bt)为 tp(A::A(bt));编译正常，因为我们显式的明确的告诉系统应该使用 A 类的转换构造函数处理，所以，显式的告诉计算机应该如何处理数据，通常可以解决多义性问题。

33.C++运算符重载赋值运算符

自定义类的**赋值运算符重载函数**的作用与内置赋值运算符的作用类似，但是要要注意的是，它与拷贝构造函数与析构函数一样，要注意深拷贝浅拷贝的问题，在没有深拷贝浅拷贝的情况下，如果没有指定默认的赋值运算符重载函数，那么系统将会自动提供一个赋值运算符重载函数。

赋值运算符重载函数的定义与其它运算符重载函数的定义是差不多的。

下面我们以实例说明如何使用它，代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Internet
{
  public:
    Internet(char *name,char *url)
    {
      Internet::name = new char[strlen(name)+1];
      Internet::url = new char[strlen(url)+1];
      if(name)
      {
        strcpy(Internet::name,name);
      }
      if(url)
      {
        strcpy(Internet::url,url);
      }
    }
    Internet(Internet &temp)
    {
      Internet::name=new char[strlen(temp.name)+1];
      Internet::url=new char[strlen(temp.url)+1];
      if(name)
      {
        strcpy(Internet::name,temp.name);
```

```
        }
        if(url)
        {
            strcpy(Internet::url,temp.url);
        }
    }
    ~Internet()
    {
        delete[] name;
        delete[] url;
    }
    Internet& operator =(Internet &temp)//赋值运算符重载函数
    {
        delete[] this->name;
        delete[] this->url;
        this->name = new char[strlen(temp.name)+1];
        this->url = new char[strlen(temp.url)+1];
        if(this->name)
        {
            strcpy(this->name,temp.name);
        }
        if(this->url)
        {
            strcpy(this->url,temp.url);
        }
        return *this;
    }
  public:
    char *name;
    char *url;
};
int main()
{
  Internet a("中国软件开发实验室","www.cndev-lab.com");
  Internet b = a;//b 对象还不存在，所以调用拷贝构造函数，进行构造处理。
  cout<<b.name<<endl<<b.url<<endl;
  Internet c("美国在线","www.aol.com");
  b = c;//b 对象已经存在，所以系统选择赋值运算符重载函数处理。
  cout<<b.name<<endl<<b.url<<endl;
  system("pause");
}
```

上例代码中的 Internet& operator =(Internet &temp)就是赋值运算符重载函数的定义，内部需要先 delete 的指针就是涉及深拷贝问题的地方，由于 b 对象已经构造过，name 和 url 指针的范围已经确定，所以在复制新内容进去之前必须把堆区清除，区域的过大和过

小都不好，所以跟在后面重新分配堆区大小,而后进行复制工作。

在类对象还未存在的情况下，赋值过程是通过拷贝构造函数进行构造处理(代码中的 Internet b = a;就是这种情况)，但当对象已经存在，那么赋值过程就是通过赋值运算符重载函数处理(例子中的 b = c;就属于此种情况)。

34.C++的 iostream 标准库介绍(1)

我们从一开始就一直在利用 C++的输入输出在做着各种练习，输入输出是由 iostream 库提供的，所以讨论此标准库是有必要的，它与 C 语言的 stdio 库不同，它从一开始就是用多重继承与虚拟继承实现的面向对象的层次结构，作为一个 c++的标准库组件提供给程序员使用。

iostream 为内置类型类型对象提供了输入输出支持，同时也支持文件的输入输出，类的设计者可以通过对 iostream 库的扩展，来支持自定义类型的输入输出操作。

为什么说要扩展才能提供支持呢？我们来一个示例。

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

class Test
{
   public:
      Test(int a=0,int b=0)
      {
         Test::a=a;
         Test::b=b;
      }
   int a;
   int b;
};
int main()
{
   Test t(100,50);
   printf("%???",t);//不明确的输出格式
   scanf("%???",t);//不明确的输入格式
   cout<<t<<endl;//同样不够明确
   cin>>t;//同样不够明确
   system("pause");
}
```

由于自定义类的特殊性，在上面的代码中，无论你使用 c 风格的输入输出，或者是 c++的输入输出都不是不明确的一个表示，由于 c 语言没有运算符重载机制，导致 stdio 库的不可扩充性，让我们无法让 printf()和 scanf()支持对自定义类对象的扩充识别，而 c++是可以通过运算符重载机制扩充 iostream 库的，使系统能能够识别自定义类型，从而让输入输出明确的知道他们该干什么，格式是什么。

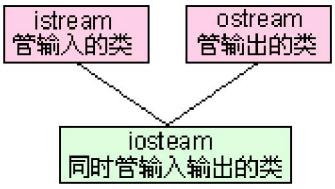
在上例中我们之所以用 printf 与 cout 进行对比目的是为了告诉大家，C 与 C++处理输入输出的根本不同，我们从 c 远的输入输出可以很明显看出是函数调用方式，而 c++的则是对象模式，**cout 和 cin 是 ostream 类和 istream 类的对象**。

C++中的 iostream 库主要包含下图所示的几个头文件：

iostream 库	
<fstream>	<iomainip>
<ios>	<iosfwd>
<iostream>	<istream>
<ostream>	<sstream>
<streambuf>	<strstream>

我们所熟悉的输入输出操作分别是由 istream(输入流)和 ostream(输出流)这两个类提供的，为了允许双向的输入 / 输出，由 istream 和 ostream 派生出了 iostream 类。

类的继承关系见下图：



iostream 库定义了以下三个标准流对象：

1.cin，表示标准输入(standard input)的 istream 类对象。cin 使我们可以从设备读如数据。
2.cout，表示标准输出(standard output)的 ostream 类对象。cout 使我们可以向设备输出或者写数据。
3.cerr，表示标准错误(standard error)的 osttream 类对象。cerr 是导出程序错误消息的地方，它只能允许向屏幕设备写数据。

输出主要由重载的左移操作符（<<）来完成，输入主要由重载的右移操作符(>>)完成。

>>a 表示将数据放入 a 对象中。
<<a 表示将 a 对象中存储的数据拿出。

这些标准的流对象都有默认的所对应的设备，见下表：

C++对象名	设备名称	C中的标准设备名	默认含义
cin	键盘	stdin	标准输入
cout	显示器屏幕	stdout	标准输出
cerr	显示器屏幕	stderr	标准错误输出

图中的意思表明 cin 对象的默认输入设备是键盘，cout 对象的默认输出设备是显示器屏幕。

那么原理上C++有是如何利用 cin / cout 对象与左移和右移运算符重载来实现输入输出的呢？

下面我们以输出为例，说明其**实现原理**：

cout 是 ostream 类的对象，因为它所指向的是标准设备（显示器屏幕），所以它在 iostream 头文件中作为全局对象进行定义。

```
ostream cout(stdout);//其默认指向的C中的标准设备名，作为其构造函数的参数使用。
```

在 iostream.h 头文件中，ostream 类对应每个基本数据类型都有其友元函数对左移操作符进行了友元函数的重载。

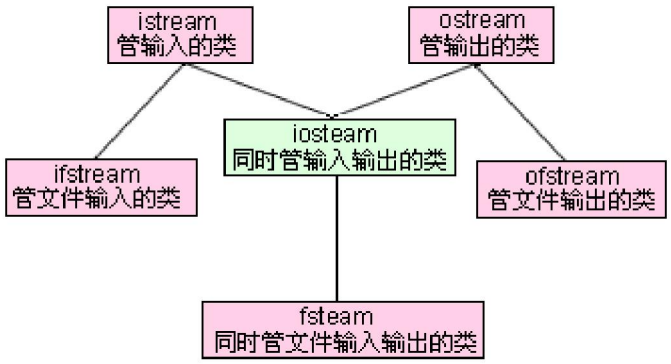
```
ostream& operator<<(ostream &temp,int source);
ostream& operator<<(ostream &temp,char *ps);
```

。。。。等等

一句输出语句：cout<<"www.cndev-lab.com"; ，事实上调用的就是 ostream& operator<<(ostream &temp,char *ps);这个运算符重载函数，由于返回的是流对象的引用，引用可以作为左值使用，所以当程序中有类似 cout<<"www.cndev-lab.com"<<"中国软件开发实验室";这样的语句出现的时候，就能够构成连续输出。

由于 iostream 库不光支持对象的输入输出，同时也支持文件流的输入输出，所以在详细讲解左移与右移运算符重载只前，我们有必要先对文件的输入输出以及输入输出的控制符有所了解。

和文件有关系的输入输出类主要在 fstream.h 这个头文件中被定义，在这个头文件中主要被定义了三个类，由这三个类控制对文件的各种输入输出操作，他们分别是 ifstream、ofstream、fstream，其中 fstream 类是由 iostream 类派生而来，他们之间的继承关系见下图所示。



由于文件设备并不像显示器屏幕与键盘那样是标准默认设备，所以它在 fstream.h 头文件中是没有像 cout 那样预先定义的全局对象，所以我们必须自己定义一个该类的对象，我们要以文件作为设备向文件输出信息(也就是向文件写数据)，那么就应该使用 ofstream 类。

ofstream 类的默认构造函数原形为：

```
ofstream::ofstream(const char *filename,int mode = ios::out,int openprot = filebuf::openprot);
```

filename:　　要打开的文件名
mode:　　要打开文件的方式
prot:　　打开文件的属性

其中 mode 和 openprot 这两个参数的可选项表见下表：

mode 属性表

ios::app:	以追加的方式打开文件
ios::ate:	文件打开后定位到文件尾，ios:app 就包含有此属性
ios::binary:	以二进制方式打开文件，缺省的方式是文本方式。两种方式的区别见前文
ios::in:	文件以输入方式打开
ios::out:	文件以输出方式打开
ios::trunc:	如果文件存在，把文件长度设为 0

可以用“或”把以上属性连接起来，如 ios::out|ios::binary。

openprot 属性表:

0：普通文件，打开访问
1：只读文件
2：隐含文件
4：系统文件

可以用“或”或者“+”把以上属性连接起来 ，如 3 或 1|2 就是以只读和隐含属性打开文件。

示例代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <fstream>
using namespace std;

int main()
{
   ofstream myfile("c:\\1.txt",ios::out|ios::trunc,0);
   myfile<<"中国软件开发实验室"<<endl<<"网址： "<<"www.cndev-lab.com";
   myfile.close()
   system("pause");
}
```

文件使用完后可以使用 close 成员函数关闭文件。

ios::app 为追加模式，在使用追加模式的时候同时进行文件状态的判断是一个比较好的习惯。

示例如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   ofstream myfile("c:\\1.txt",ios::app,0);
   if(!myfile)//或者写成 myfile.fail()
   {
      cout<<"文件打开失败，目标文件状态可能为只读！";
      system("pause");
      exit(1);
   }
   myfile<<"中国软件开发实验室"<<endl<<"网址："<<"www.cndev-lab.com"<<endl;
   myfile.close();
}
```

在定义 ifstream 和 ofstream 类对象的时候，我们也可以不指定文件。以后可以通过成员函数 open()显式的把一个文件连接到一个类对象上。

例如：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   ofstream myfile;
   myfile.open("c:\\1.txt",ios::out|ios::app,0);
   if(!myfile)//或者写成 myfile.fail()
   {
      cout<<"文件创建失败,磁盘不可写或者文件为只读!";
      system("pause");
      exit(1);
   }
   myfile<<"中国软件开发实验室"<<endl<<"网址："<<"www.cndev-lab.com"<<endl;
   myfile.close();
```

```
}
```

下面我们来看一下是如何利用 ifstream 类对象，将文件中的数据读取出来，然后再输出到标准设备中的例子。

代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
   ifstream myfile;
   myfile.open("c:\\1.txt",ios::in,0);
   if(!myfile)
   {
      cout<<"文件读错误";
      system("pause");
      exit(1);
   }
   char ch;
   string content;
   while(myfile.get(ch))
   {
      content+=ch;
      cout.put(ch);//cout<<ch;这么写也是可以的
   }
   myfile.close();
   cout<<content;
   system("pause");
}
```

上例中，我们利用成员函数 get()，逐一的读取文件中的有效字符，再利用 put()成员函数，将文件中的数据通过循环逐一输出到标准设备(屏幕)上，get()成员函数会在文件读到默尾的时候返回假值，所以我们可以利用它的这个特性作为 while 循环的终止条件，我们同时也在上例中引入了 C++风格的字符串类型 string，在循环读取的时候逐一保存到 content 中，要使用 string 类型，必须包含 string.h 的头文件。

我们在简单介绍过 ofstream 类和 ifstream 类后，我们再来看一下 fstream 类，fstream 类是由 iostream 派生而来，fstream 类对象可以同对文件进行读写操作。

示例代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   fstream myfile;
   myfile.open("c:\\1.txt",ios::out|ios::app,0);
   if(!myfile)
   {
      cout<<"文件写错误,文件属性可能为只读!"<<endl;
      system("pause");
      exit(1);
   }
   myfile<<"中国软件开发实验室"<<endl<<"网址："<<"www.cndev-lab.com"<<endl;
   myfile.close();

   myfile.open("c:\\1.txt",ios::in,0);
   if(!myfile)
   {
      cout<<"文件读错误,文件可能丢失!"<<endl;
      system("pause");
      exit(1);
   }
   char ch;
   while(myfile.get(ch))
   {
      cout.put(ch);
   }
   myfile.close();
   system("pause");
}
```

由于 fstream 类可以对文件同时进行读写操作，所以对它的对象进行初始话的时候一定要显式的指定 mode 和 openprot 参数。

接下来我们来学习一下串流类的基础知识，什么叫**串流类**?

简单的理解就是能够控制字符串类型对象进行输入输出的类，C++不光可以支持 C++风格的字符串流控制，还可以支持 C 风格的字符串流控制。

我们先看看 C++是如何对 C 风格的字符串流进行控制的，C 中的字符串其实也就是字符数组，字符数组内的数据在内存中的位置的

排列是连续的，我们通常用 char str[size]或者 char *str 的方式声明创建 C 风格字符数组，为了能让字符数组作为设备并提供输入输出操作，C++引入了 ostrstream、istrstream、strstream 这三个类，要使用他们创建对象就必须包含 strstream.h 头文件。

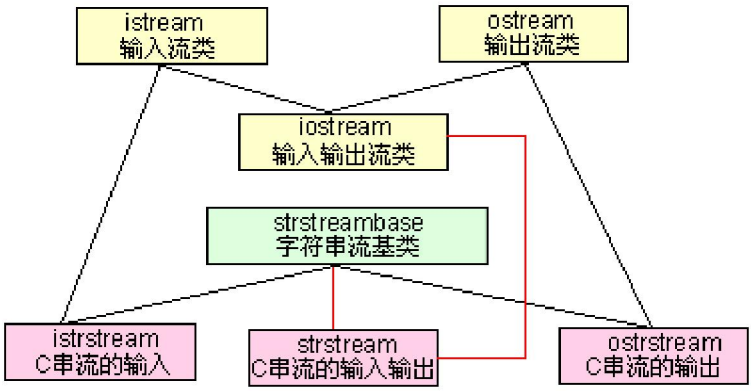
istrstream 类用于执行 C 风格的串流的输入操作，也就是以字符串数组作为输入设备。

ostrstream 类用于执行 C 风格的串流的输出操作，也就是一字符串数组作为输出设备。

strstream 类同时可以支持 C 风格的串流的输入输出操作。

istrstream 类是从 istream（输入流类）和 strstreambase（字符串流基类）派生而来，ostrstream 是从 ostream（输出流类）和 strstreambase（字符串流基类）派生而来，strstream 则是从 iostream(输入输出流类)和和 strstreambase（字符串流基类）派生而来。

他们的继承关系如下图所示：



串流同样不是标准设备，不会有预先定义好的全局对象，所以不能直接操作，需要通过构造函数创建对象。

类 istrstream 的构造函数原形如下：

istrstream::istrstream(const char *str,int size);

参数 1 表示字符串数组,而参数 2 表示数组大小，当 size 为 0 时，表示 istrstream 类对象直接连接到由 str 所指向的内存空间并以\0 结尾的字符串。

下面的示例代码就是利用 istrstream 类创建类对象，制定流输入设备为字符串数组，通过它向一个字符型对象输入数据。

代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <strstream>
using namespace std;
int main()
{
   char *name = "www.cndev-lab.com";
   int arraysize = strlen(name)+1;
   istrstream is(name,arraysize);
```

```
    char temp;
    is>>temp;
    cout<<temp;
    system("pause");
}
```

类 ostrstream 用于执行串流的输出，它的构造函数如下所示：

```
ostrstream::ostrstream(char *_Ptr,int streamsize,int Mode = ios::out);
```

第一个参数是字符数组，第二个是说明数组的大小，第三个参数是指打开方式。

我们来一个示例代码：

```
#include <iostream>
#include <strstream>
using namespace std;
int main()
{
    int arraysize=1;
    char *pbuffer=new char[arraysize];
    ostrstream ostr(pbuffer,arraysize,ios::out);
    ostr<<arraysize<<ends;//使用 ostrstream 输出到流对象的时候,要用 ends 结束字符串
    cout<<pbuffer;
    delete[] pbuffer;
    system("pause");
}
```

上面的代码中，我们创建一个 c 风格的串流输出对象 ostr，我们将 arraysize 内的数据成功的以字符串的形式输出到了 ostr 对象所指向的 pbuffer 指针的堆空间中，pbuffer 也正是我们要输出的字符串数组，在结尾要使用 ends 结束字符串，如果不这么做就有溢出的危险。

35.C++的 iostream 标准库介绍(2)

接下来我们继续看一下 **C++风格的串流控制**，C++引入了 ostringstream、istringstream、stringstream 这三个类，要使用他们创建对象就必须包含 sstream.h 头文件。

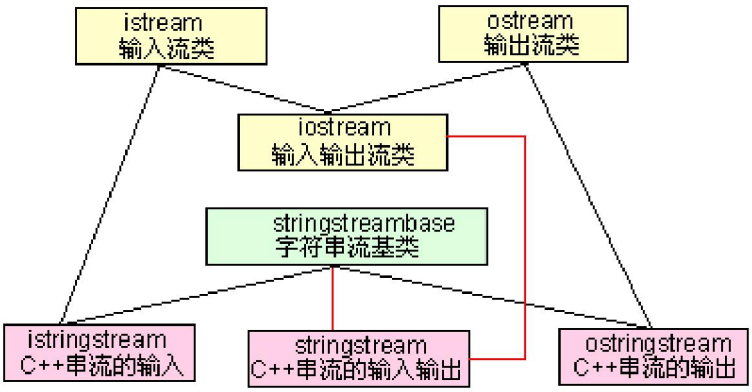
istringstream 类用于执行 C++风格的串流的输入操作。

stringstream 类同时可以支持 C++风格的串流的输入输出操作。

strstream 类同时可以支持 C 风格的串流的输入输出操作。

istringstream 类是从 istream（输入流类）和 stringstreambase（c++字符串流基类）派生而来，ostringstream 是从 ostream（输出流类）和 stringstreambase（c++字符串流基类）派生而来，stringstream 则是从 iostream(输入输出流类)和和 stringstreambase（c++字符串流基类）派生而来。

他们的继承关系如下图所示：



istringstream 是由一个 string 对象构造而来，istringstream 类从一个 string 对象读取字符。

istringstream 的构造函数原形如下：

istringstream::istringstream(string str);

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <sstream>
using namespace std;
int main()
{
istringstream istr;
istr.str("1 56.7",);
//上述两个过程可以简单写成 istringstream istr("1 56.7");
```

```
cout << istr.str()<<endl;
int a;
float b;
istr>>a;
cout<<a<<endl;
istr>>b;
cout<<b<<endl;
system("pause");
}
```

上例中，构造字符串流的时候，空格会成为字符串参数的内部分界，例子中对 a,b 对象的输入"赋值"操作证明了这一点，字符串的空格成为了整型数据与浮点型数据的分解点，利用分界获取的方法我们事实上完成了字符串到整型对象与浮点型对象的拆分转换过程。

str()成员函数的使用可以让 istringstream 对象返回一个 string 字符串（例如本例中的输出操作(cout<<istr.str();）。

ostringstream 同样是由一个 string 对象构造而来，ostringstream 类向一个 string 插入字符。
ostringstream 的构造函数原形如下：
ostringstream::ostringstream(string str);

示例代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
ostringstream ostr;
//ostr.str("abc");//如果构造的时候设置了字符串参数,那么增长操作的时候不会从结尾开始增加,而是修改原有数据,超出的部分增长
ostr.put('d');
ostr.put('e');
ostr<<"fg";

string gstr = ostr.str();
cout<<gstr;
system("pause");
}
```

在上例代码中，我们通过 put()或者左移操作符可以不断向 ostr 插入单个字符或者是字符串，通过 str()函数返回增长过后的完整字符串数据，但值得注意的一点是，当构造的时候对象内已经存在字符串数据的时候，那么增长操作的时候不会从结尾开始增加,而是修

改原有数据,超出的部分增长。

对于 stringstream 了来说，不用我多说，大家也已经知道它是用于 C++风格的字符串的输入输出的。
stringstream 的构造函数原形如下：

stringstream::stringstream(string str);

示例代码如下:

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main()
{
   stringstream ostr("ccc");
   ostr.put('d');
   ostr.put('e');
   ostr<<"fg";
   string gstr = ostr.str();
   cout<<gstr<<endl;

   char a;
   ostr>>a;
   cout<<a

   system("pause");
}
```

除此而外，stringstream 类的对象我们还常用它进行 string 与各种内置类型数据之间的转换。

示例代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
```

```
int main()
{
    stringstream sstr;
//--------int 转 string----------
    int a=100;
    string str;
    sstr<<a;
    sstr>>str;
    cout<<str<<endl;
//--------string 转 char[]--------
    sstr.clear();//如果你想通过使用同一 stringstream 对象实现多种类型的转换，请注意在每一次转换之后都必须调用 clear()成员函数。
    string name = "colinguan";
    char cname[200];
    sstr<<name;
    sstr>>cname;
    cout<<cname;
    system("pause");
}
```

接下来我们来学习一下**输入/输出的状态标志**的相关知识，C++中负责的输入/输出的系统包括了关于每一个输入/输出操作的结果的记录信息。这些当前的状态信息被包含在 io_state 类型的对象中。io_state 是一个枚举类型（就像 open_mode 一样），以下便是它包含的值。

goodbit 无错误

Eofbit 已到达文件尾

failbit 非致命的输入/输出错误，可挽回

badbit 致命的输入/输出错误,无法挽回

有两种方法可以获得输入/输出的状态信息。一种方法是通过调用 rdstate()函数，它将返回当前状态的错误标记。例如，假如没有任何错误，则 rdstate()会返回 goodbit.

下例示例，表示出了 rdstate()的用法：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

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

```
int main()
{
   int a;
   cin>>a;
   cout<<cin.rdstate()<<endl;
   if(cin.rdstate() == ios::goodbit)
   {
      cout<<"输入数据的类型正确，无错误！"<<endl;
   }
   if(cin.rdstate() == ios_base::failbit)
   {
      cout<<"输入数据类型错误，非致命错误，可清除输入缓冲区挽回！"<<endl;
   }
   system("pause");
}
```

另一种方法则是使用下面任何一个函数来检测相应的输入/输出状态：

```
bool bad();

bool eof();

bool fail();

bool good();
```

下例示例，表示出了上面各成员函数的用法：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

int main()
{
   int a;
   cin>>a;
   cout<<cin.rdstate()<<endl;
   if(cin.good())
   {
      cout<<"输入数据的类型正确，无错误！"<<endl;
   }
```

```
    if(cin.fail())
    {
        cout<<"输入数据类型错误，非致命错误，可清除输入缓冲区挽回！"<<endl;
    }
    system("pause");
}
```

如果错误发生，那么流状态既被标记为错误，你必须清除这些错误状态，以使你的程序能正确适当地继续运行。要清除错误状态，需使用 clear()函数。此函数带一个参数，它是你将要设为当前状态的标志值。，只要将 ios::goodbit 作为实参。

示例代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

int main()
{
    int a;
    cin>>a;
    cout<<cin.rdstate()<<endl;
    cin.clear(ios::goodbit);
    cout<<cin.rdstate()<<endl;
    system("pause");
}
```

通常当我们发现输入有错又需要改正的时候，使用 clear()更改标记为正确后，同时也需要使用 get()成员函数清除输入缓冲区，以达到重复输入的目的。

示例代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
using namespace std;

int main()
{
    int a;
```

```
    while(1)
    {
        cin>>a;
        if(!cin)//条件可改写为 cin.fail()
        {
            cout<<"输入有错!请重新输入"<<endl;
            cin.clear();
            cin.get();
        }
        else
        {
            cout<<a;
            break;
        }
    }
    system("pause");
}
```

最后再给出一个对文件流错误标记处理的例子，巩固学习，代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream myfile("c:\\1.txt",ios_base::in,0);
    if(myfile.fail())
    {
        cout<<"文件读取失败或指定文件不存在!"<<endl;
    }
    else
    {
        char ch;
        while(myfile.get(ch))
        {
            cout<<ch;
        }
        if(myfile.eof())
        {
            cout<<"文件内容已经全部读完"<<endl;
```

```
        }
        while(myfile.get(ch))
        {
            cout<<ch;
        }
    }
    system("pause");
}
```

36.C++的 iostream 标准库介绍(3)

C 语言提供了**格式化输入输出**的方法，C++也同样，但是 C++的**控制符**使用起来更为简单方便，在 c++下有两中方法控制格式化输入输出。

1．有流对象的成员函数。

例如，下列程序以成员函数的方式控制输出的精度：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

int main()
{
   float pi=3.14159f;
   cout<<pi<<endl;
   cout.precision(2);
   cout<<pi<<endl;
   system("pause");
}
```

2．使用 C++输入输出控制符，控制符是在拖文件 iomanip.h 中定义的对象，与成员函数有一样的效果，控制符不必像成员函数学那样单独调用，它可以直接插入流中使用。

例如，下列程序以控制符的方式控制输出的精度：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

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

int main()
{
   float pi=3.14159f;
   cout<<pi<<endl;
   cout<<setprecision(4);
```

```
    cout<<pi<<endl;
    system("pause");
}
```

下表我们列出了一些比较常用的控制符号，由于篇幅有限读者请根据自己的需要查阅相关书籍：

对于 iostream 标准库来说包含了众多的成员函数，各函数都有其自身的作用，篇幅问题笔者在这里不能一一说明例举，由于标准输入对象 cin 提供输入的时候会自动以空格作为分界，给我们获取一行带有空格的完整字符串带来了困难，在这里补充一个非常用有的成员函数－－－－getline()。

其函数原型为：

```
getlin(chiar *str,int size,char='\n');
```

第一个参数是字符数组，用于存放整行文本，第二个参数读取的最大字符个数，第三个参数为作为分界界限的字符，默认识是\n，换行符。

示例代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

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

int main()
{
    char str[100];
    cin.getline(str,sizeof(str),'\n');
    cout<<str<<endl;
    system("pause");
}
```

通过上面内容的学习，我们对 i/o 有了一些基本点基本的认识，现在是该切入正题的时候了，详细学习一下，如何重载左移与右移操作符。

先说**左移（<<）操作符**，也就是我们**常说的输出操作符**。

对于自定义类来说，重载左移操作符的方法我们常使用类的友元方式进行操作。

示例代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
   public:
      Test(int age = 0,char *name = "\0")
      {
         Test::age = age;
         strcpy(Test::name,name);
      }
      void outmembers(ostream &out)
      {
         out<<"Age： "<<age<<endl<<"Name： "<<this->name<<endl;
      }
      friend ostream& operator <<(ostream& ,Test&);
   protected:
      int age;
      char name[50];
};
ostream& operator <<(ostream& out,Test &temp)
{
   temp.outmembers(out);
   return out;
}
int main()
{
   Test a(24,"管宁");
   cout<<a;
   system("pause");
}
```

上例代码中，我们对 void outmembers(ostream &out)的参数使用 ostream 定义主要是为了可以向它传递任何 ostream 类对象不光是 cout 也可以是 ofstrem 或者是 ostrstream 和 ostringstream 类对象，做到通用性。

重载运算符，我们知道可以是非成员方式也可以是成员方式的，对于<<来说同样也可以是成员方式，但我十分不推荐这么做，因为对于类的成员函数来说，第一个参数始终是会被隐藏的，而且一定是当前类类型的。

下面的示例代码就是将上面的<<重载函数修改成成员方式的做法：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

#include <iostream>
using namespace std;

class Test
{
   public:
      Test(int age = 0,char *name = "\0")
      {
         Test::age = age;
         strcpy(Test::name,name);
      }
      void outmembers(ostream &out)
      {
         out<<"Age： "<<age<<endl<<"Name： "<<this->name<<endl;
      }
      ostream& operator <<(ostream &out)
      {
         this->outmembers(out);
         return out;
      }
   protected:
      int age;
      char name[50];
};
int main()
{
   Test a(24,"管宁");
   a<<cout;
   system("pause");
}
```

从代码实现上，我们将函数修改成了 ostream& operator <<(ostream &out)，迫不得已将 ostream 类型的引用参数放到了后面，这是因为，成员方式运算符重载函数第一个参数会被隐藏，而且一定是当前类类型的，这和 ostream 类型冲突了。由此我们在使用 cout 输出的时候就必须写成 a<<cout;，这样一来代码的可读行就大大降低了，这到底是左移还是右移呢？为此我再一次说明，对于左移和右移运算符的重载是十分不推荐使用成员函数的方式编写的。

为了巩固学习，下面我们以 fstream 对象输出为例做一个练习。

代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
#include <iostream>
#include <fstream>
using namespace std;

class Test
{
   public:
      Test(int age = 0,char *name = "\0")
      {
         Test::age = age;
         strcpy(Test::name,name);
      }
      void outmembers(ostream &out)
      {
         out<<"Age："<<age<<endl<<"Name："<<this->name<<endl;
      }
      friend ostream& operator <<(ostream& ,Test&);
   protected:
      int age;
      char name[50];
};
ostream& operator <<(ostream& out,Test &temp)
{
   temp.outmembers(out);
   return out;
}
int main()
{
   Test a(24,"管宁");
   ofstream myfile("c:\\1.txt",ios::out,0);
   if (myfile.rdstate() == ios_base::goodbit)
   {
      myfile<<a;
      cout<<"文件创建成功，写入正常！"<<endl;
   }
   if (myfile.rdstate() == ios_base::badbit)
   {
      cout<<"文件创建失败，磁盘错误！"<<endl;
   }
   system("pause");
```

```
}
```

对于左移运算符重载函数来说，由于不推荐使用成员方式，那么使用非成员方式在类有多重继承的情况下，就不能使用虚函数进行左移运算符重载的区分，为了达到能够区分显示的目的，给每个类分别**添加不同的虚函数**是必要的。

示例代码如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者

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

class Student
{
   public:
      Student(int age = 0,char *name = "\0")
      {
         Student::age = age;
         strcpy(Student::name,name);
      }
      virtual void outmembers(ostream &out) = 0;
      friend ostream& operator <<(ostream& ,Student&);
   protected:
      int age;
      char name[50];
};
ostream& operator <<(ostream& out,Student &temp)
{
   temp.outmembers(out);
   return out;
}
class Academician:public Student
{
   public:
      Academician(int age = 0,char *name = "\0",char *speciality="\0"):Student(age,name)
      {
         strcpy(Academician::speciality,speciality);
      }
      virtual void outmembers(ostream &out)
      {
         out<<"Age： "<<age<<endl<<"Name： "<<name<<endl<<
```

```
        "speciality："<<speciality<<endl;
    }
  protected:
    char speciality[80];
};
class GraduateStudent:public Academician
{
  public:
    GraduateStudent(int age = 0,char *name = "\0",char *speciality="\0",
     char *investigate="\0"):Academician(age,name,speciality)
    {
      strcpy(GraduateStudent::investigate,investigate);
    }
    virtual void outmembers(ostream &out)
    {
      out<<"Age："<<age<<endl<<"Name："<<name<<endl<<
       "speciality："<<speciality<<endl<<"investigate："<<investigate<<endl;
    }
  protected:
    char investigate[100];
};
int main()
{
  Academician a(24,"管宁","Computer Science");
  cout<<a;
  GraduateStudent b(24,"严燕玲","Computer Science","GIS System");
  cout<<b;
  system("pause");
}
```

在上面的代码中为了能够区分输出 a 对象与 b 对象，我们用虚函数的方式重载了继承类 Academician 与多重继承类 GraduateStudent 的 outmembers 成员函数，由于 ostream& operator <<(ostream& out,Student &temp) 运算符重载函数是 Student 基类的，Student &temp 参数通过虚函数的定义可以适应不同派生类对象，所以在其内部调用 temp.outmembers(out)；系统可识别不同继类的 outmembers()成员函数。

最后看一下，右移运算符的重载，右移运算符我们也常叫它输入运算符号，对于它来说，具体实现和左移运算符的重载差别并不大，对于有多成员对象的类来说，只要保证能够完整输入各成员对象大数据就可以了。

示例如下：

```
//程序作者:管宁
//站点:www.cndev-lab.com
//所有稿件均有版权,如要转载,请务必著名出处和作者
```

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

class Test
{
   public:
      Test(int age = 0,char *name = "\0")
      {
         Test::age = age;
         strcpy(Test::name,name);
      }
      void inputmembers(istream &out)
      {
         cout<<"please input age：";
         cin>>Test::age;
         cout<<"please input name：";
         cin>>Test::name;
      }
      friend istream& operator >>(istream& ,Test&);
   public:
      int age;
      char name[50];
};
istream& operator >>(istream& input,Test &temp)
{
   temp.inputmembers(input);
   return input;
}
int main()
{
   Test a;
   cin>>a;
   cout<<a.age<<"|"<<a.name<<endl;
   system("pause");
}
```

37.Eclipse3.06 + MinGW3.1 配置标准 C/C++开发环境

前言

学习 c 语言和 c++的人首先需要的是一个可提供练习的开发平台，对于 c++来说，可供使用的工具平台有很多，包括 Borland 的 c++ builder 和 Microsoft 的 Visual Studio 系列，的确他们都是非常不错的。

难道我们除了这些我们所熟悉的开发工具就没有其他的选择了吗？

对于我们此文的主角，Eclipse 来说，很多人都知道它是为 JAVA 开发而生的，但今天我们要说的是如何利用它与 MinGW 配合，设置出高效的 c++开发平台。

软件准备

1.Eclipse 3.06

官方站点：http://www.eclipse.org

Eclipse 工具下载地址：http://www.eclipse.org/downloads/index.php

2.C/C++ Development Toolkit

下载地址：http://update.eclipse.org/tools/cdt/releases/new/

注意，在这里我们选择的是 cdt-2.1.0-win32.x86

3.MinGW 3.1

下载地址：http://prdownloads.sf.net/mingw/MinGW-3.1.0-1.exe?download

系统环境

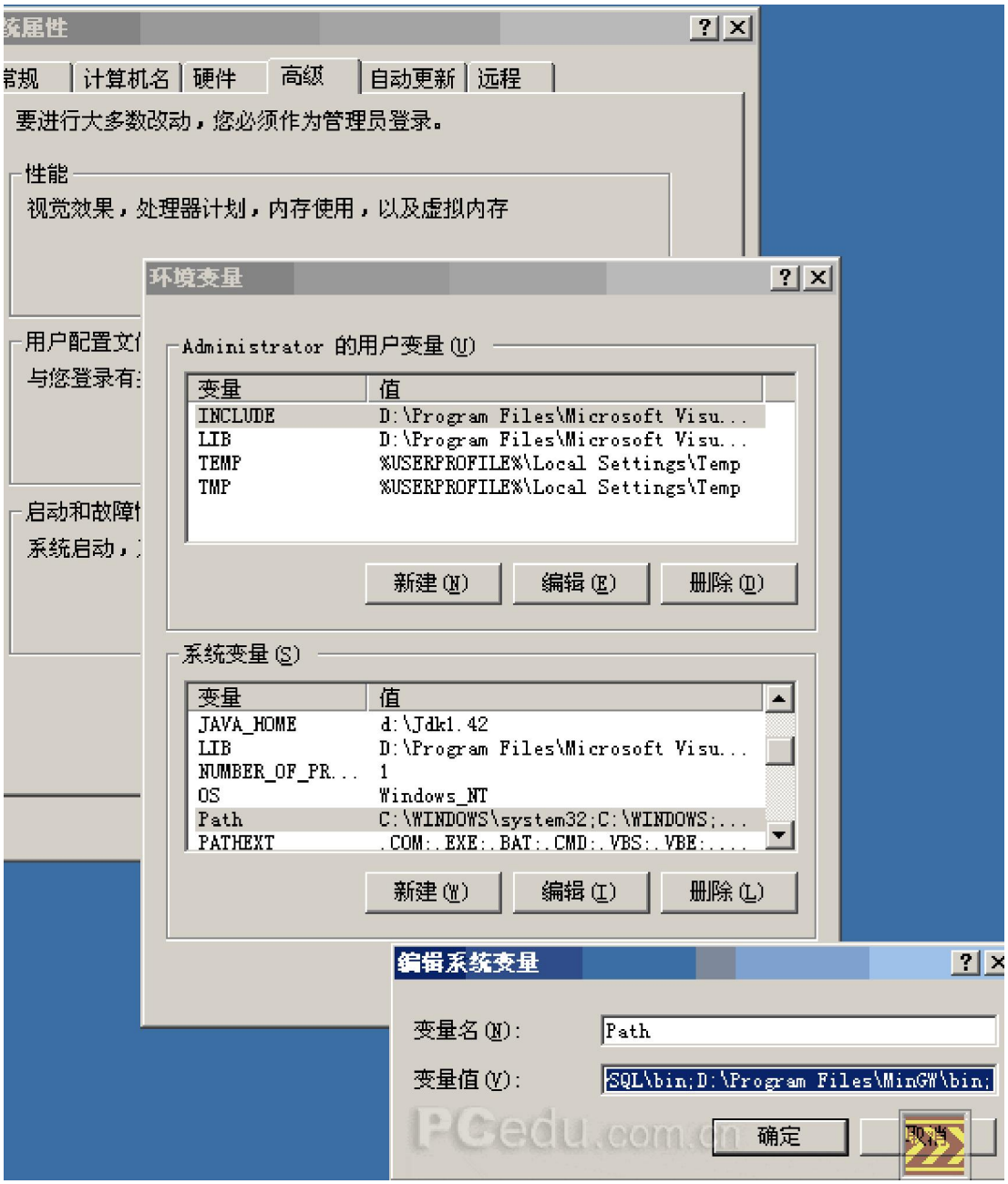
WIN2000/WINXP/WIN2003

环境配置

我们把需要的工具都下载完毕后，首先安装 MinGW，安装完毕后我们先配置一下系统的环境变量。

点击我的电脑的属性--->高级--->环境变量--->系统变量--->path

在 path 后增加:MinGW 安装路径\MinGW\bin;，如下图所示。

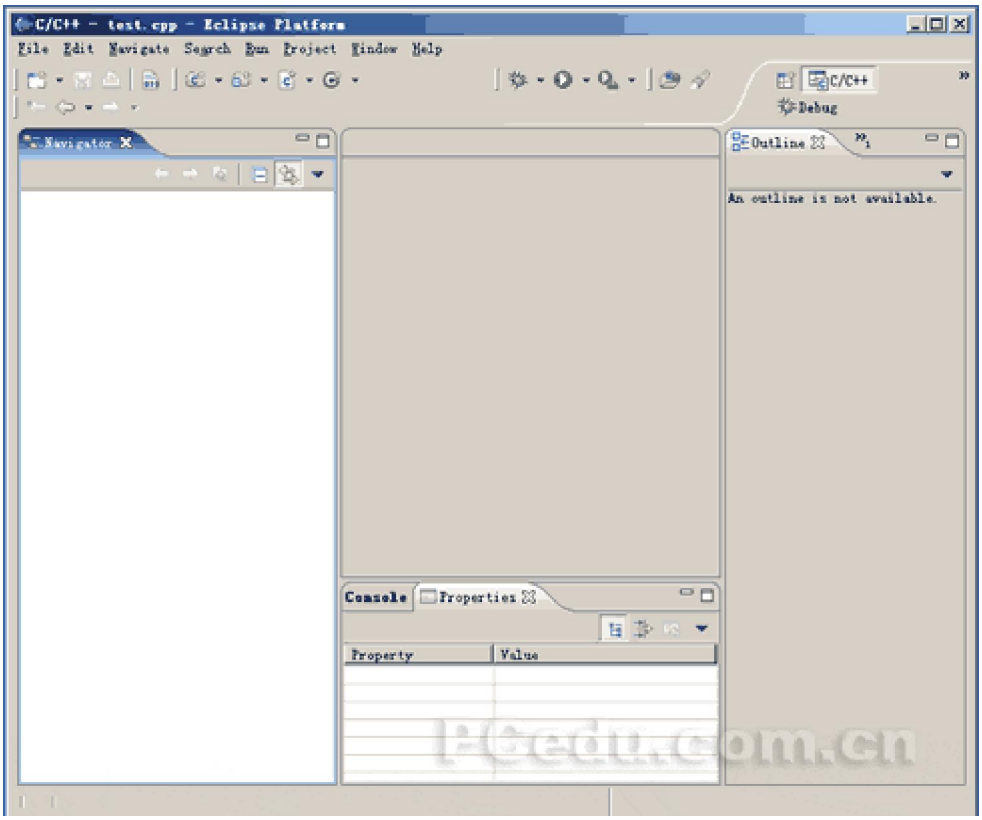


接下来找到你安装 MinGW 的目录,将 MinGW\bin\mingw32-make.exe 这个文件，改名为 make.exe。

接下来，我们安装 Eclipse。

在安装完成后先不急运行 Eclipse，将下载的 C/C++ Development Toolkit 解压后的 features、plugins 目录放到 Eclipse 的安装目录下。

至此平台配置基本完成，下面我们运行 Eclipse，第一次运行的时候系统会提示用户设置工作目录既 workspace，这也就是工程文件的安放位置，这里我们选择默认识，运行后的 Eclipse 环境界面如下图。



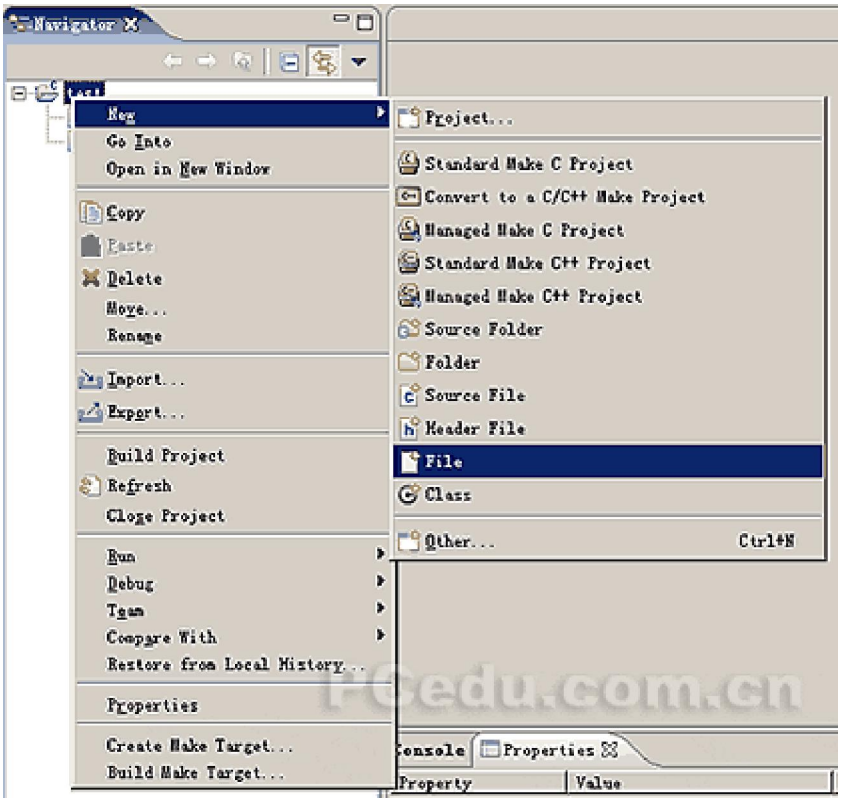
下面我们来建一个工程

选择 FILE--->NEW--->Project...

在弹出的对话框中选择 Standard Make C++ Progject，如下图所示。

之后选择 NEXT 输入工程文件名，这里我们输入 test，完成工程设置。

接下来我们展开左边的 Navigator 对话框的 test 工程目录，在空白处点击 Mouse 右键，选择 NEW-->FILE，如下图所示。



接下来，在弹出的对话框中输入要新添加的 c++文件名，这里我们输入 Hello.cpp。

下来我们在编辑对话框中输入 c++源程序,代码如下：

```
#include <iostream>

using namespace std;

int main()
{
   cout<<"你好 Eclipse!\n";
   system("pause");
}
```

为了能够使 Eclipse 环境能够速编译源文件，我们还要设置编译控制（Make Targets）。

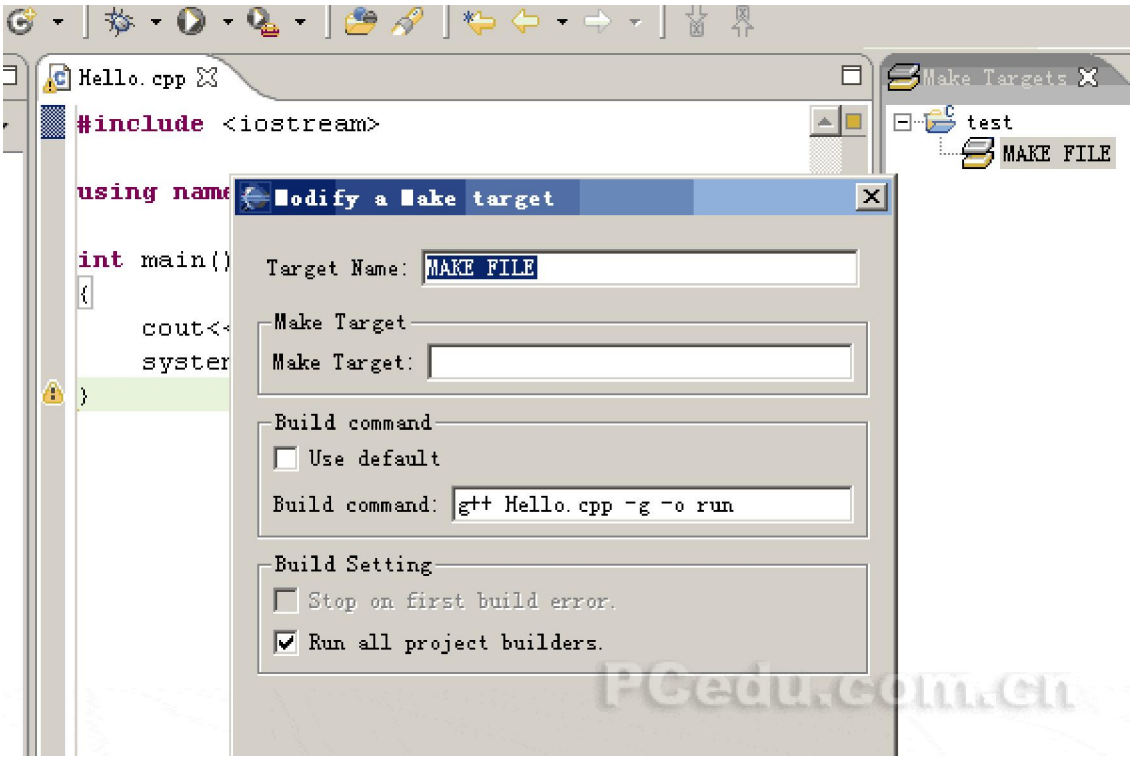
我们选择 Eclipse 上部分导航条上的 Window->Show View->Make Targets

在右边出现的 Make Targets 对话框中选择 test 工程名，Mouse 右键盘选择 Add Make Targets，在接下来弹出的对话框中输入

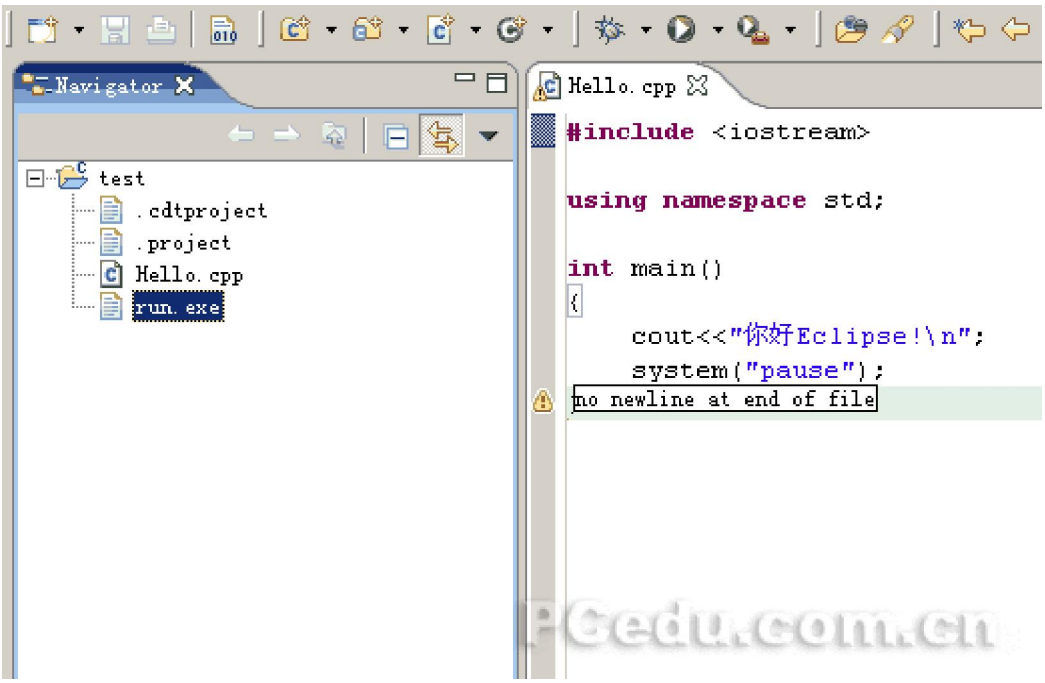
```
Target Name:MAKE FILE
Builder Command:g++ Hello.cpp -g -o run,这里如果是 c 环境就改成 gcc Hello.c -g -o run
```

完成后点击 Create。

如下图所示：



完成后，点击 Make Targets 对话框中的 MAKE FILE 分支就可以进行源文件的编译工作了，编译完成后在 Navigator 对话框中会出现 RUN.EXE，这就是编译后的可执行文件，双击就可以运行它，如下图所示意。



到此 Eclipse+MinGW 的标准 c/c++环境就已经全部配置并测试完毕，读者如果还有问题可在讨论区提问，我会给予解答。